

آموزش زبان C#

http://csharp-persian.netfirms.com/Lesson_14_Delegates_Events_In_C_Sharp.htm

مطالبی که در این قسمت خواهید دید :

شیء چیست؟

ارث‌بری یا Inheritance

انتزاع یا Abstraction

کیسوله کردن یا Encapsulation

چندریختی یا Polymorphism

منابع این مطلب

مفاهیم شیء گرایی در C#

یکی از مهمترین و اصلی ترین مشکلات برای افرادی که به تازگی با محیط‌های برنامه نویسی شیء‌گرا آشنا می شوند، درک مفاهیم شیء‌گرایی است. در حقیقت درک مفاهیمی چون شیء و مولفه (Component) بسیار دشوار نیست، کافایت کمی به اطراف خود با دقت نگاه کنیم. ما در دنیایی از اشیاء مختلف زندگی میکنیم. تلویزیون، رادیو در و پنجره، همه و همه نمونه هایی از اشیاء مختلفی هستند که در اطراف ما وجود دارند. اما درک و پیاده‌سازی این مفهوم در یک زبان برنامه‌سازی اندکی متفاوت است.

شیء چیست؟

همانطور که گفتیم، با یک دید تصویری به سادگی می توانید اشیاء مختلفی را در اطراف خود ببینید. تمامی این اشیاء دارای سطوح و درجه پیچیدگی متفاوتی هستند. پیچیدگی آنها به شکل ظاهری و نوع رفتار آنها بستگی دارد. در شیء گرایی به "شکل ظاهر" در اصطلاح، صفت یا Attribute و به عملی که شیء انجام می دهد، رفتار یا Behavior می گویند.

برای مثال یک صندلی را در نظر بگیرید. صندلی صفات مختلفی دارد ولی رفتار خاصی ندارد. مثلاً پایه های صندلی جزو صفات آن بشمار می روند. با کمی دقیق تر شدن می توان از این صفات برای توصیف صندلی استفاده کرد. بعنوان مثال تعداد پایه های صندلی می تواند عددی بین 3 تا 5 باشد. محل نشستن صندلی می تواند جمله‌ای در وصف جنس آن و مقدار مصرف ماده سازنده آن باشد. پستی صندلی را نیز می توان بعنوان متغیری boolean در نظر گرفت چراکه برخی از صندلی ها فاقد پستی هستند. با استفاده از این سه صفت ساده، به راحتی می توان صندلی را توصیف نمود و با همین سه صفت میتوان گونه‌های مختلفی از صندلی را نیز توصیف کرد.

منظور از رفتار، عملی است که یک شیء انجام می دهد. از اینرو برای صندلی نمی توان به سادگی صفات آن، رفتاری را متصور شد. مثلاً می‌توانیم بگوئیم تاشو بودن صندلی یکی از رفتارهای آن می تواند باشد، چراکه عملی است که می تواند یک صندلی آنرا انجام دهد.

حال شیء دیگری مانند تلویزیون را در نظر بگیرید. صفاتی که می توان برای تلویزیون در نظر گرفت عبارتند از : صفحه نمایش، سازنده آن و ... برای تلویزیون به راحتی می توان رفتار در نظر گرفت : خاموش و روشن شدن، تغییر کانال و کم

و زیاد کردن صدا. این رفتارها بر اثر درخواست یک انسان یا همان کاربر اتفاق می افتند. شیء تلویزیون را میتوان بصورت زیر نمایش داد :



بطور کلی، یک شیء را میتوان بوسیله صفات و رفتارهای آن بطور کامل توصیف نمود. یک شیء حتماً جسمی فیزیکی نیست، بلکه هر چیز قابل تصویری است که دارای صفت و رفتار است. در حقیقت میتوان گفت یک شیء شبیه به یک اسم است. اگر بتوان برای این اسم، صفت و رفتاری تعریف کرد، آن وقت تبدیل به شیء می شود. از دید انتزاعی، زمان را نیز می توان بعنوان یک شیء در نظر گرفت. صفات زمان، ساعت، دقیقه و ثانیه هستند و گذشت زمان، رفتار آن است. در ایجاد شیء هیچ محدودیتی وجود ندارد و همه چیز به تخیل شما باز میگردد. در زبان C#، اشیاء بوسیله کلاسها (Class) نمایش داده می شوند. داخل کلاس، صفات بصورت فیلدها ظاهر می شوند و جهت پیاده سازی رفتارها از متدها استفاده می گردد. به مثال زیر توجه نماید :

```
class Time
{
    int hours;
    int minutes;
    int seconds;

    void pastime()
    {
        //some implementation
    }
}
```

در این مثال، کلاس Time مشاهده می شود. این کلاس با کلمه کلیدی class اعلان گردیده است. همانطور که میدانید، دو گروه باز و بسته {} نیز ابتدا و انتهای کلاس را مشخص می کند. فیلدها دارای نام و نوع هستند. متدها دارای نام و نوع بازگشتی می باشند و پیاده سازی آنها داخل بلوک مربوط به خود آنها (بین دو {}) انجام میگردد. یک شیء می تواند هر یک از نیازمندی های یک پروژه باشد. طراحی و اعلان صحیح اشیاء و مشخص کردن صفات و رفتار آنها یکی از مقوله های مهم در مهندسی نرم افزار بر پایه شیء گرائی است، چراکه همگی تراکنشها بین اشیاء صورت می پذیرند.

کلاسه کردن اشیاء (مقدمه ای بر ارث بری Inheritance)

طبقه‌بندی اشیاء در گروه‌های مختلف بسیار سودمند است. زمین‌شناسان سنگها را طبقه‌بندی می‌کنند و زیست‌شناسان گیاهان و حیوانات را طبقه‌بندی می‌کنند. طبقه‌بندی اشیاء باعث می‌شود تا با دقت و ظرافت بیشتری بتوان به جزئیات هر طبقه و یا هر نوع پرداخت.

برای مثال حیوانات را در نظر بگیرید. زیست‌شناسان حیوانات را به دسته‌های پرندگان، مهره‌داران و خزندگان تقسیم می‌کنند. پرندگان صفاتی همچون منقار و بال دارند، مهره‌داران بدنی مودار دارند و خون گرمند و خزندگان خون سردند. از دید رفتاری نیز پرندگان پرواز می‌کنند، مهره‌داران به بچه‌های خود شیر می‌دهند و خزندگان می‌خزند. اینها ویژگیهای ارشد یا عام هستند و صفات دیگری در زیر گروهها افزوده می‌شوند. در گروه اصلی تنها صفات و رفتارهایی قرار می‌گیرند که عمومیت داشته باشند و سایر صفات و رفتارها در زیر گروهها معین می‌گردند. بعنوان مثال کلاس پرندگان را بشکل زیر میتوان پیاده‌سازی نمود :

```
class Bird
{
    string beakDescription;
    int wingSpan;
    string typeOfBird;

    void fly()
    {
        //some implementation
    }
}
```

می‌توان این کلاس را بعنوان کلاسی عمومی برای پرندگان در نظر گرفت که دارای فیلدی جهت تعیین نوع پرنده نیز می‌باشد. با استفاده از فیلد `typeOfBird` می‌توان گونه پرنده مورد نظر را معین نمود.

سلسله مراتب اشیاء (بررسی ارث بری در محیط شیء‌گرا)

روش گفته شده در بالا، جهت افزودن صفات برای یک طبقه‌بندی ساده کارآمد است، اما برای طبقه‌بندی‌های پیچیده قابل قبول نیست. تصور کنید حجم عظیمی نیازمندی در پروژه وجود داشته باشد و به سطح‌های مختلفی از طبقه‌بندی نیاز داشته باشید. با افزودن صفات جدید به هر یک از طبقات و سطوحها به پیچیدگی پروژه به شدت افزوده می‌شود.

اتفاقی که در مورد زیر شاخه‌ها رخ می‌دهد، ایجاد یک سلسله مراتب طبیعی است. برنامه‌نویسی شیء‌گرا، متدولوژیی را جهت مدیریت سلسله مراتب طبیعی فراهم می‌نماید. بعنوان مثال، اگر در یک پروژه یکی از طبقه‌بندیهای ما حیوانات باشند، منطقی است که شیءای از طبقه حیوانات در رأس سلسله مراتب قرار گیرد و در طبقات و زیر شاخه‌های سطوح پائینی، پرندگان، مهره‌داران و خزندگان قرار گیرند. در سطح بعدی می‌توان چند پرنده مانند اردک، کلاغ و ... را نیز در زیر شاخه پرندگان قرار داد. این سلسله مراتب تا سطح مورد نظر قابل تعمیم است.

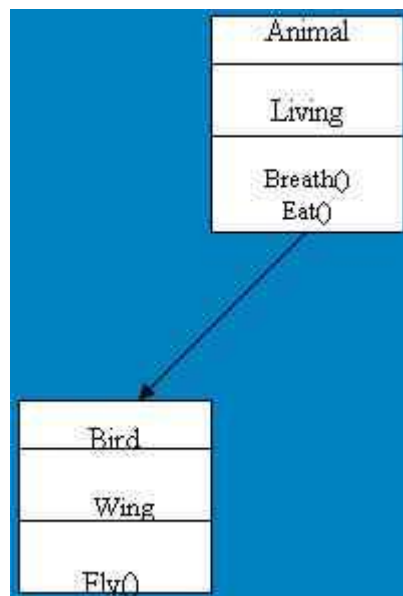
در برنامه نویسی شیء‌گرا، مفهومی که اشیاء را تحت یک سلسله مراتب خاص قرار میدهد، ارث‌بری یا **Inheritance** نامیده می‌شود. مثلاً طبقه‌بندی حیوانات را در نظر بگیرید، شیءای که در بالاترین سطح قرار می‌گیرد، شیء **Animal** است. این شیء دارای ویژگیهایی بسیار کلی و عمومی است و باید چنین هم باشد، چراکه سرگروه باید همیشه ویژگیهایی

را داشته باشد که در زیر شاخه ها بطور مشترک وجود دارد و هر یک از زیر شاخه ها به ویژگیهای سرگروه خود، ویژگیها و مشخصات خاص خود را می افزایند.

در این مثال، سرگروه **Animal** تنها می تواند دارای صفت یگانه "زیستن" باشد، چراکه همین صفت آنرا از سایر اشیاء، نظیر سنگ و در و پنجره و بطور کل اجسام، متمایز می نماید. رفتار این شیء نیز می تواند "تنفس کردن" و "غذا خوردن" باشد. حال برای اینکه شیء ای بتواند در این سلسله مراتب وارد شود، حداقل باید دارای صفت شیء سرگروه این سلسله مراتب باشد، در غیر اینصورت نمی تواند در زیر شاخه آن قرار گیرد.

تفاوت **Animal** با سایر اشیاء که در این سلسله مراتب قرار می گیرند در آنست که سایر اشیاء می بایست صفاتی دیگر و - یا رفتارهای دیگری را نیز به صفات و رفتارهای **Animal** اضافه کنند. این صفات و رفتارها مسلماً جزئی تر و دقیق تر از صفات و رفتارهای سرگروه است. همین مسئله مفهومی را در پیمایش سلسله مراتبها بوجود می آورد: در پیمایش بالا به پائین (**Top-Down**) سلسله مراتبها به جزئیات یا گونه های خاص برخورد می کنیم، حال آنکه در پیمایش پائین با بالا (**Bottom-Up**) به گروهها و دسته های عمومی می رسمیم. اشیاء سطوح پائینی (که به آنها **child** می گوئیم) صفات و رفتارهای اشیاء سطح بالاتر خود را به ارث می برند. به این اشیاء بالاتر یا سرگروهها نیز **parent** می گوئیم. به این رابطه موجود بین **child** و **parent** در اصطلاح رابطه "هست" یا "بودن" (**is-a relationship**) می گویند. مثلاً می گوئیم "اردک یک پرند است".

سادگی ارث بری از نحوه ایجاد سلسله مراتب ارث بری نشأت می گیرد. اشیاء سطوح پائینی (**child**) در تعریف خود اشیاء سطوح بالایی (**parent**) خود را مشخص می کنند. در این جا تنها کاری که لازم است یک **child** نسبت به **parent** خود انجام دهد افزودن صفات و رفتارهای مربوط به خود است. شکل زیر یک سلسله مراتب را نشان می دهد.



همانطور که در این شکل ملاحظه می کنید، سرگروه که همان **Animal** است، دارای صفت **Living** و رفتارهایی همچون **Eat()** می باشد. زیر شاخه این شیء کلیه صفات آنرا به ارث می برد، بعلاوه اینکه صفت **Wing** و رفتار **Fly()** را نیز به

صفات و رفتارهای **parent** خود می‌افزاید. حال برای زیر شاخه‌ای که در زیر **Bird** قرار می‌گیرد نیز می‌توان صفات و رفتارهای خاص دیگری در نظر گرفت..

علاوه بر طبقه‌بندی اشیاء در دسته‌های مختلف و سادگی در سازماندهی آنها بوسیله ارث‌بری، استفاده از ارث‌بری در انجام کارها نیز صرفه‌جویی ایجاد می‌کند. هر شیء جدیدی که به یک سلسله مراتب وارد می‌شود، بطور خودکار تمامی صفات و رفتارهای کلیه **parent** های خود را دارا می‌باشد و بعثت ارث‌بری نیازی به تعریف مجدد این صفات برای شیء جدید نمی‌باشد. به بیان دیگر می‌توان گفت، ارث‌بری روشی برای استفاده مجدد از صفات و رفتارهای موجود است. استفاده از ارث‌بری این امکان را برای طراحان نرم افزار فراهم می‌کند تا وقت بیشتری برای تفکر بر روی منطق برنامه صرف کنند و درگیر پیچیدگی‌های پیاده‌سازی و نگهداری نرم افزار نشوند.

انتزاع (Abstraction)

اکنون زمان مناسبی برای بحث درباره انتزاع است. برخی اشیاء تا حدودی انتزاعی هستند و برخی دیگر کاملاً واقعی. بعنوان مثال، چیزی بعنوان **Animal** وجود ندارد، بلکه این تنها توصیف کلاسی از اشیاء است. همچنین موجودیتی وجود ندارد که از لحاظ فیزیکی یک پرنده باشد. این تنها طبقه‌بندی و دسته‌بندی است که مورد استفاده قرار می‌گیرد. از طرف دیگر شیء‌ای وجود دارد بنام اردک که واقعاً یک اردک است و دارای کلیه صفات و رفتارهای آن می‌باشد. البته باید توجه داشت که هر اردک تنها نمونه‌ای از "اردک" است. (منظور در اینجا آنست که موجودیتی مانند یک اردک که واقعا وجود دارد، خود نمونه‌ای (**instance**) از کلاس اردک است. درک مطالب در اینجا شاید به اندکی تأمل و حوصله نیاز داشته باشد!)

برای شیء‌ای مانند **Animal** یا **Bird**، نمی‌توان صفتی همچون "پاهایی پهن" و یا رفتاری مانند "را رفتن شبیه به اردک" را تعریف نمود. همچنین از اشیاء **Animal** و **Bird** تنها یک نمونه می‌تواند وجود داشته باشد که این نمونه‌ها نیز بسیار مهم هستند، چراکه این نمونه‌ها هستند که ساختار یک سلسله مراتب را تشکیل می‌دهند و صفات و رفتارهای کلی آن را معین می‌نمایند. (البته توجه کنید که استفاده از کلمه **abstract** در زبان **C#** خود یگانه بودن و همچنین انتزاعی بودن کلاس را نشان می‌دهد.)

تعریف و توصیف صحیح این اشیاء انتزاعی، همچون **Animal** و **Bird**، در طبقه‌بندی و کارآمدی ساختار سلسله مراتبی بسیار می‌تواند موثر باشد. مثال زیر نحوه تعریف و ایجاد اشیاء انتزاعی در **C#** را نشان می‌دهد.

```
abstract class Animal
{
    //abstract definitions and implementations
}

class Bird : Animal
{
    //class implementation
}
```

در این مثال کلاس **Animal** بصورت **abstract** اعلان شده تا مشخص شود که شیء ای انتزاعی است. چون شیء انتزاعی عملاً وجود ندارد، نمی‌توان نمونه‌ای جدید از روی آن ایجاد کرد و تنها یک شیء از آن وجود دارد. در کلاس دوم، **Bird**، نشان داده شده است که این کلاس از کلاس انتزاعی **Animal** ارث‌بری دارد. این عمل با استفاده از ":" در جلوی نام **Bird** و سپس به دنبال آن نام کلاسی که **Bird** از آن ارث‌بری می‌کند، یعنی همان **Animal**، صورت گرفته است.

اشیاء درون اشیاء (مقدمه ای بر کپسوله کردن یا Encapsulation)

ساختارهای سلسله مراتبی روشی جهت دستیابی به روابط بین اشیاء هستند. هر چند روشهای دیگری نیز برای نشان دادن روابط بین اشیاء وجود دارد. یکی از این روشها که بسیار معمول نیز هست، استفاده از اشیاء درون اشیائی دیگر است. یکی از اصطلاحات رایج برای بیان این مفهوم، کپسوله کردن یا **Encapsulation** است. استفاده صحیح از این مفهوم باعث کاهش پیچیدگی شده و تنها آن قسمت اطلاعات و جزئیات را نمایان می‌کند که برای درک یک شیء لازم است. بعنوان مثال، پرواز پرندگان مسئله‌ای است که توجه بسیاری از زیست شناسان را به خود جلب کرده است. آنها برای درک پرواز، ساختار بالها، پرها و اسکلت پرندگان را مورد مطالعه قرار می‌دهند. برای درک پرواز، بررسی همین اجزاء و جزئیات کافیهست و نیازی به مطالعه درباره سیستم گوارش پرندگان نمی‌باشد. برای بررسی یک موضوع تنها اطلاعاتی مربوط به آن در حل و بررسی مسئله می‌تواند موثر باشد و اطلاعات اضافی، نه تنها وقت گیر و بی فایده است، بلکه می‌تواند گیج کننده نیز باشد و بر پیچیدگی مسئله، بدون دلیل، بیفزاید. با استفاده از کپسوله کردن تنها آن قسمت از اطلاعات که مورد نظر است در دسترس قرار می‌گیرند.

شیء **Bird** را که قبلاً درباره آن صحبت کردیم، در نظر بگیرید. پرندگان دارای منقار و بال هستند. خود بال یک شیء است که دارای صفاتی نظیر پر و اندازه است. رفتار آن نیز می‌توان باز و بسته شدن در حین پرواز باشد. نکته قابل اشاره در اینجا، کلمه "داشتن" است. اعمالی وجود دارند که پرنده آنها را انجام می‌دهد و این اعمال خود جزئی از پرنده هستند. بطور کلی، رابطه "داشتن" (**has-a relationship**) بین یک شیء و اعمالی که بر روی اجزای خود انجام می‌دهد، وجود دارد. بعنوان مثال می‌گوئیم "پرنده بال دارد". در اینجا چون بال شیء ای متعلق به پرنده است که پرنده روی آن عملی انجام می‌دهد، مفهوم کپسوله کردن رخ می‌دهد، شیء ای درون شیء ای دیگر. مثال زیر نحوه پیاده‌سازی کپسوله کردن را نشان می‌دهد.

```
class Wing
{
    int foreWingSize;
    int backWingSize;

    void flap()
    {
        //implementation
    }

    void fold()
    {
```



```

        //implementation
    }
}

class Bird : Animal
{
    int beakSize;
    Wing wings;

    void Fly()
    {
        //implementation
    }
}

```

در این مثال، دو کلاس **Bird** و **Wing** وجود دارند. کلاس **Wing** فیلدها و متدهای مربوط به خود را دارد. درون کلاس **Bird** اعلانی از کلاس **Wing** با استفاده از نام **wings** صورت گرفته است. این عمل رابطه مالکیتی بین **Bird** و **Wing** ایجاد می‌نماید. "پرندۀ دارای بال است". تنها چیزی که کافیسیت بدانیم آنست که کلاس انتزاعی پرندۀ دارای بال است. توجه کنید که صفات و رفتارهای کلاس **Wing** بوسیله خود این کلاس کنترل می‌شوند.

اشیائی با رفتارهایی متفاوت (مقدمه ای بر چندریختی یا Polymorphism)

در برخی شرایط ممکن است نیاز داشته باشیم تا علاوه بر اینکه شیءای را در رده خاصی طبقه‌بندی می‌کنیم، به آن شیء این اجازه را نیز بدهیم تا بتواند رفتارهای خاص به خود را داشته باشد.

برای مثال، اردک و یا عقاب اشیایی هستند که با گروه **Bird** ارتباط دارند. تصور کنید زیست‌شناسی می‌خواهد رفتارهای این پرندگان را بررسی نماید. برای بررسی این پرندگان لازم است تا زیست‌شناس آنها را در جایی نگهداری کرده و بر روی آنها مطالعه انجام دهد. مسلماً این پرندگان باید در قفس نگهداری شوند اما مطمئناً قفس اردک با قفس عقاب یکسان نمی‌تواند باشد. این بدین معناست، قفسی را که تعریف می‌کنیم باید طوری باشد تا برای اکثر گونه‌های پرندگان مناسب باشد. می‌توانیم قفسی مخصوص پرندگانی که پرواز می‌کنند در نظر بگیریم. از اینرو پرندگانی را که پرواز می‌کنند را در این قفس قرار می‌دهیم. همانطور که مشاهده می‌کنید، دسته‌بندی پرندگان در یک گروه جهت رسیدن به اهداف پروژه ضروری است. پیاده‌سازی مطلب گفته شده بشکل زیر است.

```

abstract class flyingBird : Bird
{
    //implementation
}

class Eagle : flyingBird
{
    //implementation
}

class Duck : flyingBird
{
    ۸

```

```
//implementation
}

class Experiment
{
    public static void Main()
    {
        flyingBird flyingBirdCage = new flyingBird[2];
        flyingBirdCage[0] = new Eagle();
        flyingBirdCage[1] = new Duck();
    }
}
```

در کد فوق، 3 کلاس جدید ایجاد شده است. اولین کلاس، **flyingBird** است که از کلاس **Bird** مشتق شده است. از آنجائیکه تمامی پرندگان پرواز نمی‌کنند، این کلاس می‌تواند تنها مخصوص آن دسته از پرندگانی باشد که می‌توانند پرواز نمایند. کلاسهای **Eagle** و **Duck** در این کد از **FlyingBird** مشتق شده‌اند. آخرین کلاس نیز **Experiment** است که متد **Main()** در آن قرار گرفته است. درون متد **Main()**، آرایه‌ای بنام **flyingBirdCage** از نوع **FlyingBird** تعریف شده که این آرایه، مخصوص اشیایی از نوع **FlyingBird** است. از آنجائیکه کلاسهای **Duck** و **Eagle** از نوع **FlyingBird** هستند، پس می‌توان آنها را در این آرایه قرار داد. نکته مهم در اینجا نیز همین مطلب است که از این پس ما پرندگانی داریم که پرواز می‌کنند و در قفس مخصوص نگهداری می‌شوند. حال تصور کنید که این نوع قفس وجود نداشت و مجبور بودیم تا برای هر پرنده نوع خاصی از قفس را تعریف نمائیم. در این شرایط حجم کار بسیار زیاد می‌شد و به مشکل بر می‌خوردیم. اما هم اکنون می‌دانیم که قفسی داریم که شرایط کلی و عمومی برای نگهداری پرندگانی که پرواز می‌کنند را داراست. در صورتیکه برای هر پرنده بخواهید قفسی مخصوص ایجاد کنید، کدی شبیه به کد زیر مورد نیاز بود:

```
class Experiment
{
    public static void Main()
    {
        Eagle eagleCage = new Eagle();
        Duck duckCahge = new Duck();
    }
}
```

علاوه بر این تصور کنید که بخواهیم مشخصات دیگری نیز به کلاسها بیفزائیم. در این شرایط کار با کدام گزینه ساده‌تر است؟ کار با آرایه‌ای از اشیاء یکسان، یا کار با اشیایی متفاوت؟ مسلماً کار با آرایه‌ها ساده‌تر است چرا که با استفاده از تنها یک دستور **foreach** می‌توان کلیه عناصر یک آرایه را پیمایش کرد.

پس از ایجاد کلاسهای مربوطه، نوبت به بررسی نحوه پرواز هر پرنده می‌رسد. اگر کلیه پرندگان مورد مطالعه را از نوع پرندگانی که پرواز می‌کنند در نظر بگیریم و آنها را در قفس مخصوص این پرندگان قرار دهیم، مسئله بعدی نحوه پرواز هر یک از این پرندگان است.

عمل پرواز کردن، در کلیه این پرندگان وجود دارد، ولی نحوه پرواز در هر یک از آنها مسلماً متفاوت است. این مفهوم دقیقاً با مفهوم چندریختی (**Polymorphism**) در شیء‌گرایی مطابقت دارد. **Polymorphism** توانایی انجام یک عمل توسط اشیاء مختلف است که به شکل‌های مختلف انجام می‌گیرد. مثال زیر نحوه پیاده‌سازی **Polymorphism** در زبان C# را نشان می‌دهد.

```
using System;

class FlyingBird
{
    public virtual void Fly()
    {
        Console.WriteLine("This shouldn't be called");
    }
}

class Eagle : FlyingBird
{
    public override void Fly()
    {
        Console.WriteLine("Eagle Fly");
    }
}

class Duck : FlyingBird
{
    public override void Fly()
    {
        Console.WriteLine("Duck Fly");
    }
}

class Experiment
{
    public static void Main()
    {
        FlyingBird[] flyingBirdCage = new FlyingBird[2];

        flyingBirdCage[0] = new Eagle();
        flyingBirdCage[1] = new Duck();

        foreach(FlyingBird bird in flyingBirdCage)
        {
            bird.Fly();
        }
    }
}
```

خروجی این برنامه بشکل زیر است :

Eagle Fly
Duck Fly

در این مثال، کلاسهای **Eagle** و **Duck** از کلاس **FlyingBird** مشتق شده‌اند. هر یک از این کلاسها دارای متد **Fly()** هستند. تنها تفاوت این متدها در نحوه اعلان و پیاده‌سازی آنهاست. در کلاس **FlyingBird** متد **Fly()** بصورت **virtual** اعلان شده است، بدین معنا که کلاسهای مشتق شده از این کلاس می‌توانند شخصاً به پیاده‌سازی این متد پردازند. پیاده‌سازی این متد در کلاسهای مشتق شده با استفاده از کلمه کلید **override** صورت می‌گیرد. در متد **Main()** با استفاده از یک حلقه **foreach** تک تک این متدها فراخوانی می‌شوند. بعلاوه اینکه کلیه این متدها بصورت **override** تعریف شده‌اند، در فراخوانی بصورت مجزا اجرا می‌شوند.

class="MsoNormal" dir="RTL" style="text-align: right;">>

مفاهیم تکنولوژی .Net

مقدمه

C# یکی از زبانهای جدید برنامه‌سازی شی‌گرا است که با ارائه رهیافت **Component-Based** به طراحی و توسعه نرم‌افزار می‌پردازد. آنچه ما در حال حاضر از زبانهای برنامه‌سازی **Component-Based** در اختیار داریم و آنچه که **C#** در اختیار ما قرار می‌دهد، افق جدیدی به سوی تولید و طراحی نرم‌افزارهای پیشرفته را در روی ما قرار می‌دهند. نرم‌افزار، به عنوان یک سرویس، هدف اصلی نسل بعدی در سیستم‌های محاسباتی است. برای مثال، **C#** زبانی مناسب برای تولید و طراحی صفحات وب، ایجاد اجزایی با قابلیت استفاده مجدد و ایجاد محیط‌هایی چند رسانه‌ای را به عنوان زبانی که هدفش توسعه ایجاد نرم‌افزارهای پیشرفته است، در اختیار ما قرار می‌دهد. زبان برنامه‌سازی **C#**، به همراه تکنولوژی جدید شرکت نرم‌افزاری مایکروسافت یعنی **.Net**، ارائه گردید، از این رو از تکنولوژی **.Net**، این شرکت بهره می‌برد. پس در ابتدا به بیان مطالبی درباره محیط **.Net** می‌پردازیم.

چرا .Net؟

در گذشته زبانهای برنامه‌سازی، سیستم‌های عامل و محیط‌های اجرایی نرم‌افزارها برای دوره‌ای خاص ساخته می‌شدند. هنگامیکه برنامه‌ها از محیط‌های رومیزی (**Desktop**) به اینترنت منتقل می‌شدند، ابزارهای موجود نیازمند **API** های اضافی و قابلیت‌های دیگری بودند. بیشتر این قابلیت‌ها در کنار زبانهای برنامه‌سازی بعنوان ابزارهایی جهت رفع این نیازمندیها ارائه می‌شدند. هرچند این ابزارهای اضافی بصورت قابل توجهی نیازمندیها را حل کرده و باعث رسیدن اینترنت به وضعیت کنونی شدند، اما همچنان مسائل بسیاری وجود داشت که نیاز به حل شدن داشتند.

Net. به منظور پشتیبانی از کاربردهای عصر جدید اینترنت ساخته شد. مواردی همچون گسترش، امنیت و **versioning**. که از مسایل مهمی بودند، توسط **Net**. پوشش داده شدند. قسمت مرکزی **Net**. بخش **Common CLR (Language Runtime)** است که یک موتور اجرایی مجازی است که از توسعه، امنیت و ارتقای نسخه کد پشتیبانی می‌نماید. در گذشته چنین امکاناتی برای کدهای کامپایل شده فراهم نبود. بدلیل اینکه **Net**. توانست بر این مشکلات اساسی فائق آید، راه حل قدرتمندتری جهت ساخت برنامه‌های تحت اینترنت به شمار می‌رود.

Net. چیست؟

Net. محیطی جهت ساخت برنامه‌های توزیع شده است که شامل ابزارهایی نظیر "کتابخانه کلاسهای پایه" (**BCL : CLR Base Class Library**) و زبانهای برنامه‌نویسی است. این ابزارها امکان ساخت انواع مختلفی از نرم‌افزارها، از قبیل فرمهای ویندوز، **ADO.Net**، **ASP.Net** و سرویسهای وب، را فراهم می‌آورند.

فرمهای ویندوز، مجموعه‌ای از کتابخانه‌ها جهت ساخت رابط‌های کاربر گرافیکی برای برنامه‌های کاربردی است. این کتابخانه‌ها اغلب **API** های **Win32** را در خود دارا می‌باشند. همچنین امکان استفاده از رهیافت شی‌گرایی را جهت تولید آسان برنامه‌های تحت ویندوز، فراهم می‌آورند.

ADO.Net مجموعه‌ای از کلاسهای شی‌گرایی است که جهت ساخت مولفه‌های داده و سطوح دسترسی داده در برنامه‌های **n-tiered** مورد استفاده قرار می‌گیرد.

ASP.Net شامل مدل برنامه‌نویسی فرمهای وب است که بوسیله آن برنامه‌های تحت وب ساخته شده و تحت اینترنت قابلیت اجرا پیدا کرده و از طریق مرورگر (**Browser**) قابل دسترسی می‌باشند. این روش مدل بهبود یافته برنامه‌سازی وب است که در آن کدها در سرور کامپایل می‌شوند ولی همانند صفحات **HTML** در کامپیوتر مشتری اجرا می‌شوند.

سرویسهای وب، رهیافتی جدید، مستقل از **platform** و استاندارد، جهت ایجاد ارتباط و فعالیت بین سیستمهای ناهمگون در اینترنت، می‌باشند. سرویسهای وب **Net**، از زیر ساخت شی‌گرایی برنامه‌نویسی **ASP.Net** استفاده می‌کنند، اما همچنان از استانداردهای باز و مدلی بر پایه پیغام (**Message Based Model**) استفاده می‌نمایند. استفاده از استانداردهای باز از قبیل **XML**، **WSDL** و **UDDI** باعث می‌شوند تا سرویسهای وب با سایر سرویسهای وب استاندارد که پیاده‌سازی‌هایی متفاوت دارند، بدون توجه به محیط و **platform** آنها، ارتباط برقرار نمایند.

این چند نمونه، اندکی از انواع مختلف نرم‌افزارهایی بودند که می‌توان تحت **Net**. به پیاده‌سازی آنها پرداخت.

کتابخانه‌های کلاس‌های پایه (**BCL : Base Class Library**)

BCL در **Net**، شامل هزاران نوع قابل استفاده، جهت افزایش بهره‌وری در ساخت برنامه‌های **Net**. است. به علت گستردگی **BCL** یادگیری تمام کلاسهای آن وقت‌گیر بوده و امکان پذیر نمی‌باشد، به همین دلیل برای صرفه‌جویی در زمان بهتر است قبل از ایجاد یک نوع خاص به جستجوی نوع‌های موجود در **BCL** بپردازیم. نگاهی کلی به **BCL** می‌تواند بسیار سودمند باشد. جدول زیر **Namespace** های مهم و توضیح نوعهای مختلف **BCL** را نمایش می‌دهد.

.NET Namespaces	
Namespace	Description

.NET Namespaces	
Namespace	Description
System	The most commonly used types.
System.CodeDom	Allows creating types that automate working with source code, that is, compilers and code creation tools.
System.Collections	Collection types such as ArrayList , Hashtable , and Stack .
System.ComponentModel	Supports building reusable components.
System.Configuration	Types for working with various kinds of XML configuration files.
System.Data	Most of the types for ADO.NET database programming. Other types are in namespaces that are specific to a database or data interface.
System.Diagnostics	Process , EventLog , and Performance Counter types.
System.DirectoryServices	Managed interface for accessing Windows Active Directory Services.
System.Drawing	GDI+ types.
System.EnterpriseServices	COM+ types.
System.Globalization	Types for culture-specific support of calendars, formatting, and languages.
System.IO	Directory , File , and Stream types.
System.Management	APIs for performing WMI tasks.
System.Messaging	Types for working with message queues.
System.Net	Access to networking protocol types.
System.Reflection	Reflection APIs for inspecting assembly metadata.
System.Resources	Types for culture-specific resource management.
System.Runtime	COM Interop, Remoting, and Serialization support.
System.Security	Code access security, role-based security, and cryptography types.
System.ServiceProcess	Types for building Windows Services.
System.Text	Text encoding/decoding, byte array from/to

.NET Namespaces	
Namespace	Description
	string translation, the <code>StringBuilder</code> class, and regular expressions.
<code>System.Timers</code>	Timer types.
<code>System.Threading</code>	Threads and synchronization types.
<code>System.Web</code>	HTTP Communications, ASP.NET, and Web Services types.
<code>System.Windows</code>	Windows Forms types.
<code>System.XML</code>	All XML support types, including XML Schema, <code>XmlTextReaders/XmlTextWriters</code> , <code>XPath</code> , XML Serialization, and XSLT.

جدول 1-1 Namespace های مهم و رایج

هر **Namespace** مجموعه‌ای از کلاسهای از پیس ساخته شده **Net**. است که می‌توان از آنها در برنامه‌های مختلف استفاده نمود.

CLR (Common Language Runtime)

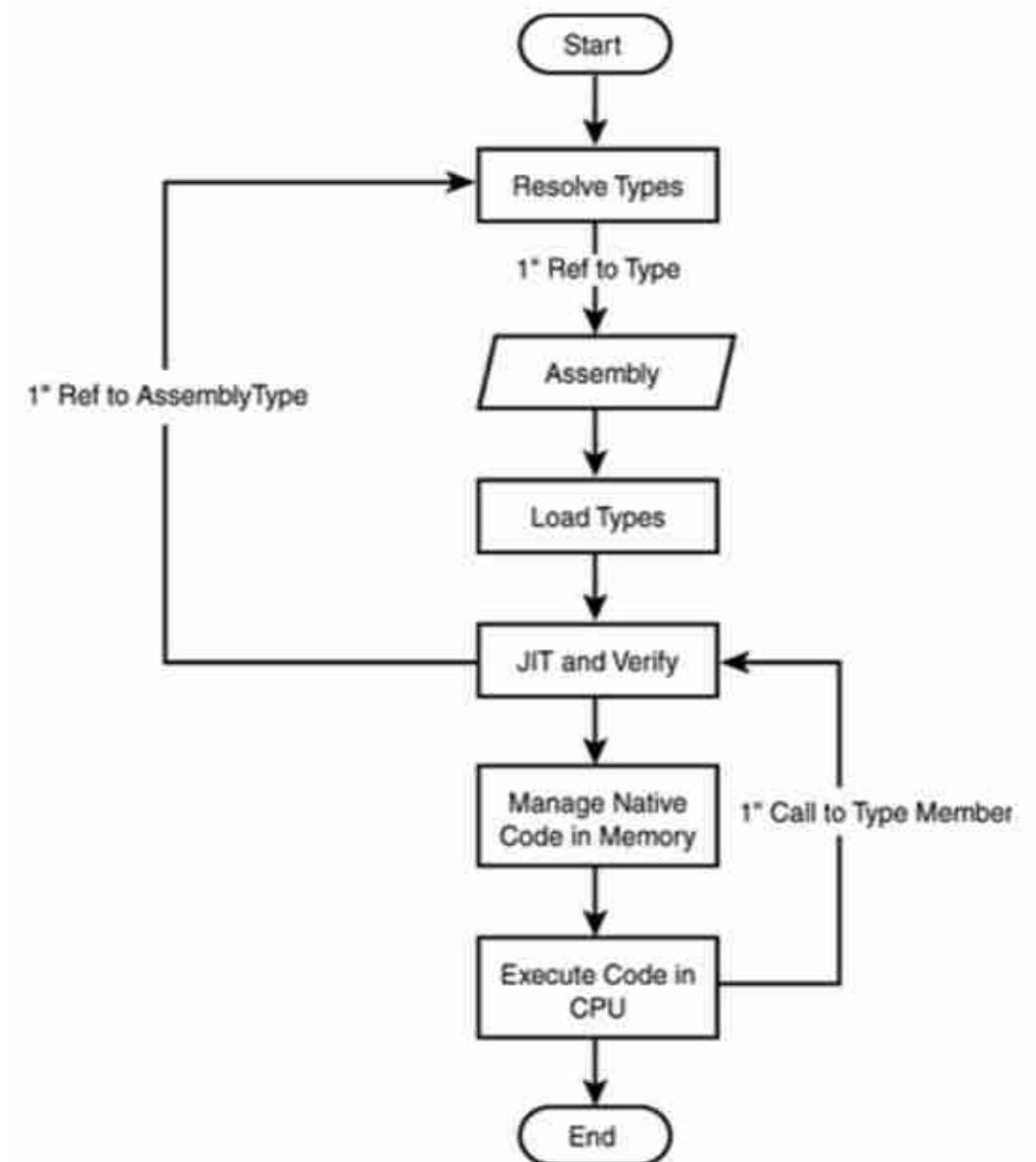
CLR یک موتور اجرایی است که با هدف اصلی اجرای هدایت شده کدها در **Net**. ایجاد گردیده است. **CLR** به مدیریت اجرا، ارتقای نسخه و امنیت تمامی کدها در **Net**. می‌پردازد. به همین دلیل کدهای **Net**. یا **C#** اغلب تحت عنوان کدهای مدیریت شده، شناخته می‌شوند. **(Managed Code)** تمامی کدهایی که به **CLR** مرتب هستند، تحت عنوان "مدیریت شده" و کدهایی توسط **CLR** مدیریت نشده‌اند، بلکه مستقیماً به کد ماشین تبدیل می‌شوند، تحت عنوان "مدیریت نشده" بیان می‌شوند.

کدهای مدیریت شده، به کد ماشین کامپایل نمی‌شوند، بلکه به زبان سطح میانی مایکروسافت (**MSIL**) کامپایل شده و مورد استفاده قرار می‌گیرند. این زبان سطح میانی را می‌توان زبانی شبیه به زبان اسمبلی تصور کرد. **IL** در حافظه بارگذاری می‌شود و بلافاصله بوسیله **CLR** در حافظه به کد ماشین کامپایل می‌گردد.

برنامه‌های **Net**. از اسمبلی‌هایی تشکیل شده‌اند که اجزای خودکار منطقی توسعه، شناسایی و امنیت به حساب می‌آیند و تفاوت آنها با روشهای قدیمی در آن است که اسمبلی می‌تواند شامل یک یا چندین فایل باشد. اسمبلی **Net**. به صورت یک فایل اجرایی تک یا یک فایل کتابخانه‌ای است، اما ممکن است حاوی ماژول‌ها، که کدهایی غیر اجرایی بوده و قابلیت استفاده مجدد را دارند، نیز باشد.

مسئله مهم دیگر در مورد **CLR**. نحوه بارگذاری (**Load**) و اجرای برنامه توسط آن است. به محض اینکه برنامه **Net**. شروع به اجرا می‌کند، ویندوز اسمبلی **Net**. را تشخیص داده و **CLR** را اجرا می‌کند. سپس **CLR** نقطه شروع برنامه را

شناسایی و پروسه تعیین انواع که در آن، محل قرارگیری انواع مختلف بکار رفته در برنامه مشخص می‌شود را، اجرا می‌کند. اسمبلی شناسایی شده در پروسه **Loader** بارگذاری می‌گردد.



شکل 1-1 نحوه مدیریت برنامه‌ها توسط CLR

زبانهای برنامه‌نویسی

قسمت مهم دیگر **Net**، پشتیبانی آن از چندین زبان برنامه‌نویسی متفاوت است. **IL** طوری طراحی شده است که از چندین زبان برنامه‌نویسی پشتیبانی نماید. در حقیقت، هم اکنون ده‌ها زبان برنامه‌نویسی مورد پشتیبانی و پذیرش **IL** می‌باشند. علاوه بر **Net**، **C#**، شامل زبانهایی نظیر **Visual Basic**، **Jscript**، **J#** و **C++** نیز می‌باشد. برخی دیگر از زبانهای برنامه‌سازی مهم که بوسیله **IL** پشتیبانی می‌شوند عبارتند از: **Cobol.Net**، **Borland Delphi.Net**، **Perl.Net**، **Python.Net**، **Fujitsu** و بسیاری دیگر از زبانهای برنامه‌سازی که تحت **Net** عمل می‌کنند و بوسیله آن مورد پذیرش هستند.

عامل اصلی نگهدارنده این زبانها در کنار یکدیگر، **CTS (Common Types System)** است. اگرچه هر یک از این زبانها نوعهای خود را در روشهای خاص خود ارائه می‌دهند، رفتار زیرساختی هر یک از آنها نسبت به **CLR** یکسان است. **CTS** اعضای یک نوع را مشخص می‌نماید: فیلد، متد، رخداد، ویژگی (**Property**) و اندیکسر (**Indexer**). همچنین سطوح دسترسی به آنها را نیز معین می‌نماید: عمومی (**public**)، داخلی (**internal**)، حفاظت شده (**protected**)، حفاظت شده داخلی (**protected internal**) و خصوصی (**private**). البته باید توجه کرد که مسلماً کلمات کلیدی هر یک از زبانها با سایر زبانها متفاوت است اما ساختار اصلی آنها در **CLR** یکسان است.

سوال مهمی که در اینجا مطرح می‌گردد، اینست که چرا از چندین زبان استفاده می‌شود؟ برای پاسخ به این سوال توجه شما را به پروژه‌های تجاری عظیم جلب می‌کنم. همانطور که می‌دانید هر پروژه تجاری دارای شرایط و ویژگیهای خاص به خود است و یکی از مهمترین عوامل در تولید و راهبری پروژه‌های امروزی امکان استفاده مجدد از برنامه‌ها است. استفاده از چندین زبان برنامه‌نویسی اولاً می‌تواند نیاز هر نوع پروژه‌ای را طبق خواسته‌های آن برآورده کند و ثانیاً قابلیت استفاده مجدد را افزایش می‌دهد.

یکی دیگر از مزایای استفاده از چندین زبان برنامه‌نویسی، تجارت بین‌المللی است، بدین معنا که هر شرکت می‌تواند نرم‌افزار و محصول خود را با یکی از زبانهای مورد نظر خود ساخته و بدون نگرانی از عدم همخوانی آن با سایر محصولات به بازار ارایه نماید. پشتیبانی **Net** از رنج وسیعی از زبانهای برنامه‌نویسی امکان به اشتراک‌گذاری کدها و استفاده مجدد از برنامه‌ها را به راحتی فراهم کرده و عصر جدیدی را در تولید نرم‌افزار ایجاد نموده است.

مزایای پشتیبانی از چندین زبان برنامه‌نویسی

این امکان باعث می‌شود تا هر فرد با توجه به علایق و سوابق کاری خود به برنامه‌نویسی با زبانی خاص بپردازد. بعنوان مثال فرض کنید گروهی مدتها با **COBOL** برنامه‌نویسی کرده‌اند، حال چون **Net** از این زبان نیز پشتیبانی می‌کند، این گروه با صرف مدت زمانی کوتاه می‌توانند به **Net** روی آورده و از مزایای آن بهره‌مند شوند. از دیگر مزایای چند زبانی استفاده مجدد از مولفه‌ها و اجزای برنامه‌های نوشته شده است.

البته به یک نکته مهم باید توجه کرد، که منظور در این جا این نیست که در یک پروژه با چندین زبان شروع به برنامه‌نویسی کنیم، اما با قابلیت **Net** می‌توانیم مثلاً **dll** های نوشته شده به زبان **C#** را در یک پروژه‌ای که با زبان **VB.Net** نوشته می‌شود، مورد استفاده قرار دهیم.

در این جا باید به این نکته نیز توجه کرد که همگونی و سازگاری بین دو زبان همیشه بصورت کامل و خودکار صورت نمی‌گیرد و در برخی موارد هر زبان ویژگی‌های خاص خود را دارد که در زبان دیگر قابل اجرا و شناسایی نمی‌باشد. بعنوان مثال، برنامه‌های **VB.Net** نمی‌توانند با **dll** های **C#** که دارای متدهای عمومی هستند و نوع آنها به طور اشاره‌گر (**pointer**) تعریف شده است، کار نمایند.

"خصوصیات عمومی زبان" یا **CLS** به منظور حل چنین مشکلاتی طراحی شده است. **CLS** ویژگی‌های عمومی یک زبان را مشخص می‌کند و تعیین می‌کند که زبانها در صورت نیاز به اشتراک‌گذاری کدها تا چه حدی می‌توانند عمل

نمایند. بعنوان مثال، **C#** برای اینکه بخواهد با **CLS** همخوانی داشته باشد، نباید اشاره‌گرها و نوع‌های بدون علامت را به صورت عمومی (**public**) در نظر بگیرد.

نصب محیط **Visual Studio.Net**

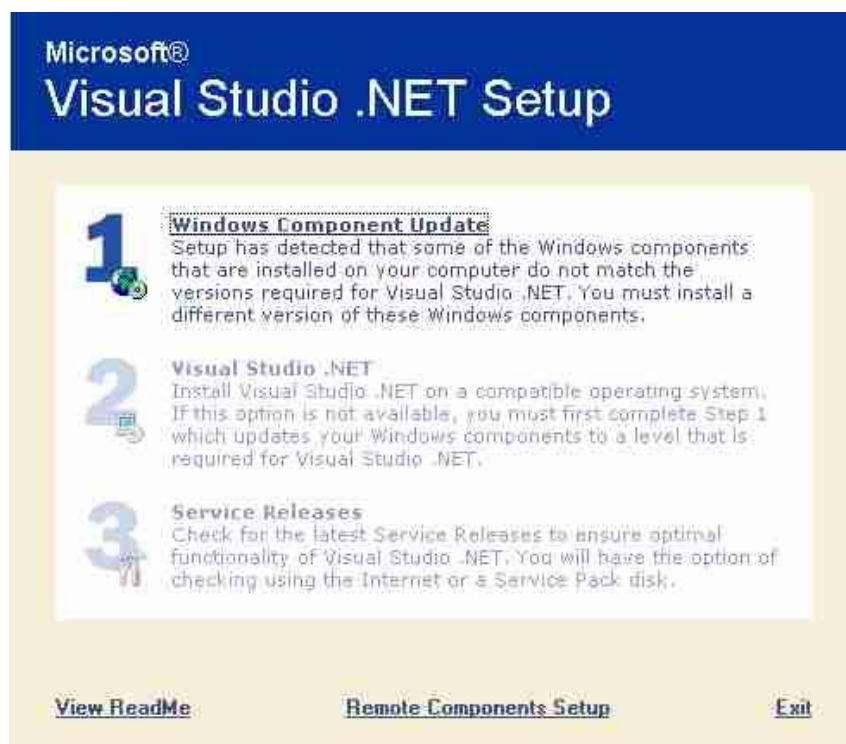
در این قسمت، به مراحل نصب **VS.Net** می‌پردازم. نصب این محیط اندکی با **C#Builder** متفاوت بوده و کمی وقت‌گیر است. در حال حاضر دو نسخه متفاوت از این محیط وجود دارد. یکی محیط قدیمی‌تر **Visual Studio.Net** و دیگری **Visual Studio.Net 2003**. البته نسخه بتای **Visual Studio.Net 2005** نیز عرضه شده است که بدلیل اینکه هنوز نسخه نهایی آن موجود نیست، از بحث در مورد آن خودداری می‌کنیم و طریقه نصب آنرا به زمانی موکول می‌کنیم که نسخه اصلی و کامل آن عرضه شود. توجه نمایید که به دلیل تفاوت‌های اندکی که در مورد نصب دو نسخه وجود دارد، مراحل نصب هر دو نسخه را توضیح خواهم داد.

نصب **Visual Studio.Net**

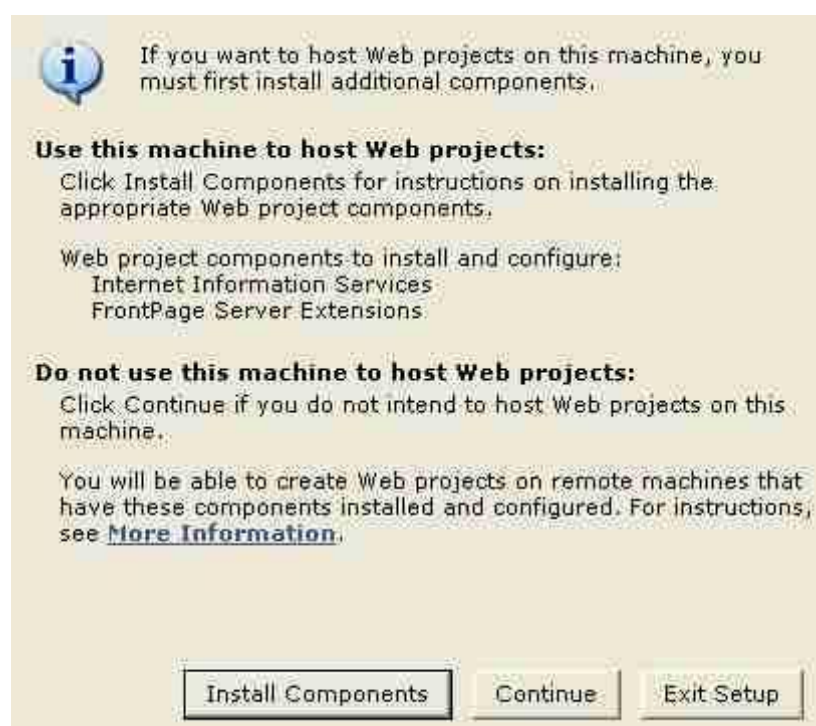
تمامی نسخه‌های مختلف **VS.Net** به دو فرم مختلف عرضه می‌شوند. فرم اول تنها شامل محیط‌های مختلف برنامه‌سازی و ابزارهای مربوط به آن است و نوع دوم علاوه بر این محیط‌ها و ابزارها، شامل فایل‌های کمکی و **Help** که به اصطلاح به آنها **MSDN** می‌گویند، می‌باشد. نسخه‌هایی که دارای **MSDN** هستند معمولاً تحت عنوان **Develop MSDN** عرضه می‌شوند. در ایران پک‌های متفاوتی از این محصول موجود است. تفاوت این پک‌ها از تعداد **CD** های آنها مشخص می‌شود. پک معمولی این نرم‌افزار حاوی سه **CD** است که تنها شامل **Net Framework 1.0** و محیط‌های برنامه‌سازی **.Net**، **C++**، **Visual Basic.Net**، **C#.Net** و **Crystal Report** است. پک دیگری در بازار ایران موجود است که حاوی 5 عدد **CD** است و علاوه بر موارد فوق حاوی **MSDN** نیز می‌باشد. اما پک اصلی و کامل این نرم‌افزار حاوی 7 عدد **CD** است که علاوه بر موارد گفته شده در بالا، حاوی **Visio For Visual Studio.Net** و **Source Safe 6.0** نیز می‌باشد. توصیه من به شما تهیه کردن پک کامل و نسخه اصلی این برنامه است، چراکه در پک کامل این محصول، ابزارها و امکانات بیشتری نیز نسبت به پک‌های دیگر تعبیه شده است. من نصب برنامه را با توجه به پک اصلی آن که حاوی 7 عدد **CD** است، توضیح خواهم داد که البته تفاوت چندانی با نصب انواع دیگر ندارد و فقط توضیحات آن بیشتر است، چراکه گزینه‌های قابل نصب در آن بیشتر هستند.

دوستان عزیز! که عضو سایت هستند و تمایل دارند تا پک کامل **Visual Studio.Net** را تهیه نمایند، می‌توانند از طریق سایت اقدام به تهیه آن کنند. برای این منظور خواهشمندم با ایمیل من مکاتبه نمایند.

برای شروع نصب، ابتدا **CD** شماره یک محصول را درون **CD-Rom** خود قرار دهید. پس از قرار دادن **CD** محصول، صفحه‌ای همانند شکل زیر نمایان خواهد شد.

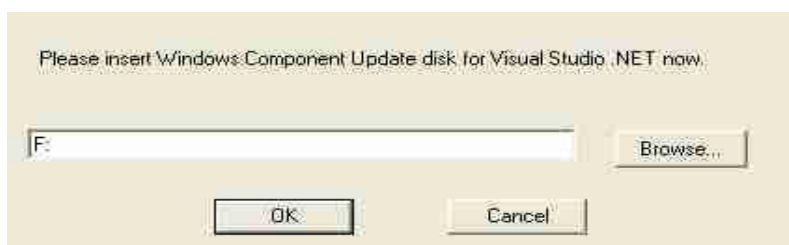


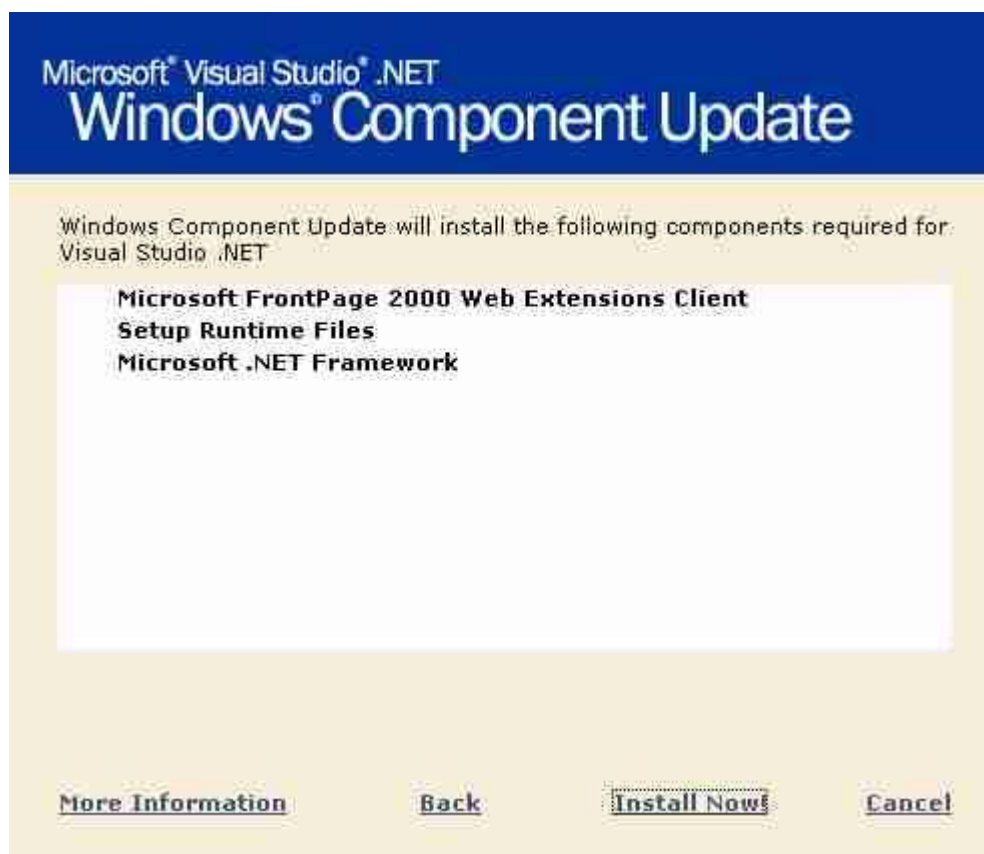
برای اجرای محیط VS.Net، مولفه‌هایی نظیر Net Framework 1.0 و FrontPage Extensions 2002 مورد نیاز است. در صورتیکه این مولفه‌ها بر روی سیستم شما نصب نشده باشند، پس از قرار دادن CD برنامه، صفحه ظاهر شده به شما نشان می‌دهد که ابتدا می‌بایست این مولفه‌ها را نصب نمایید. بر روی گزینه 1 (Windows Component Update) کلیک کنید تا این مولفه بر روی سیستم شما نصب شوند. پس از انتخاب این گزینه صفحه‌ای همانند شکل زیر ظاهر خواهد شد.



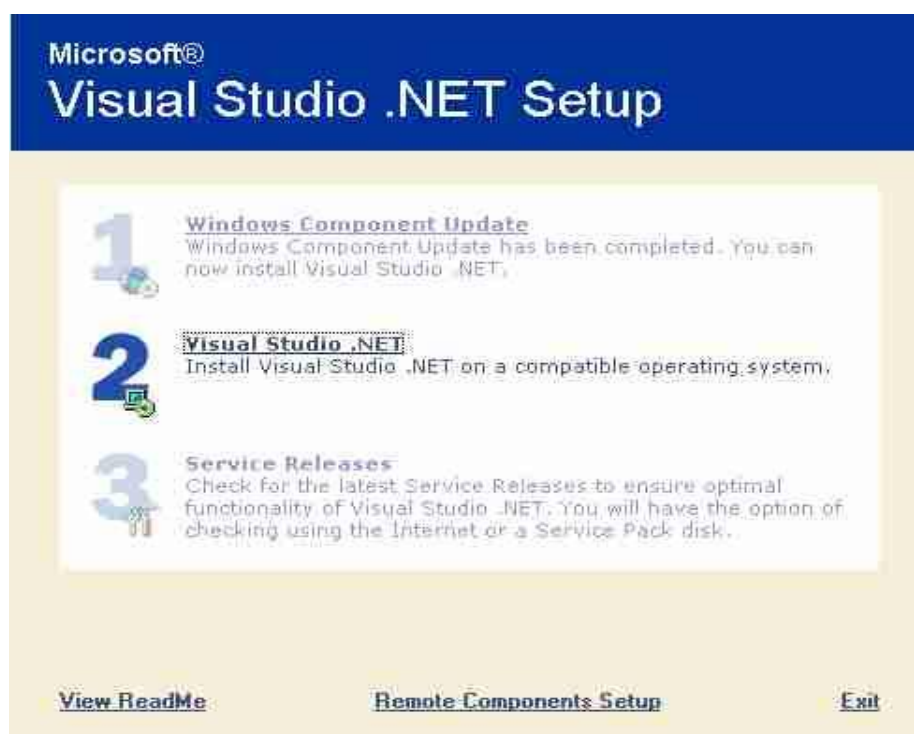
در این صفحه دو گزینه برای انتخاب وجود دارد. در صورتیکه می‌خواهید کامپیوتر شما به یک **Web Server** اجرا برنامه‌های **ASP.Net** تبدیل شود، بر روی گزینه **Install Components** کلیک نمایید. در اینصورت مولفه‌های مربوطه بر روی سیستم شما نصب خواهند شد. قبل از نصب این مولفه‌ها صفحه راهنمایی برای شما ظاهر می‌شود که اطلاعاتی درباره نحوه نصب و تنظیم این مولفه‌ها در آن آورده شده است. توجه نمایید با انتخاب این گزینه، مولفه‌هایی نظیر **IIS** و ... که مربوط به اجرای برنامه‌های **ASP.Net** هستند بر روی سیستم شما بطور خودکار نصب نخواهد شد. بلکه تنها راهنمائیهای لازم جهت نصب این مولفه‌ها و تبدیل کردن کامپیوتر خود به یک **Web Server** در اختیار شما قرار خواهد گرفت. این قسمت را پس از نصب محیط **VS.Net** نیز می‌توانید تنظیم و نصب نمایید.

توصیه من به شما، پس از ظاهر شدن این صفحه، برای نصب مولفه‌های مورد نیاز بر روی **Continue** کلیک نمایید. در صورتیکه بر روی سیستم خود دارای دو درایو **CD-ROM** هستید، **CD** مربوط به **Additional Components** را که بطور طبیعی می‌بایست **CD** شماره 5 باشد، را درون دومین درایو خود قرار دهید تا نصب این مولفه‌ها از روی آن انجام شود. در صورتی هم که تنها دارای یک درایو **CD-ROM** هستید، پس از کلیک بر روی **Continue**، صفحه باز می‌شود و از شما مسیر مولفه‌های مورد نظر را می‌خواهد، در این حالت **CD** اول را از درایو خارج نمایید و سپس **CD** مربوط به این مولفه‌ها را درون درایو خود قرار دهید و سپس گزینه **OK** را انتخاب نمایید. پس از انجام این مراحل، نصب مولفه‌های مورد نیاز آغاز خواهند شد.

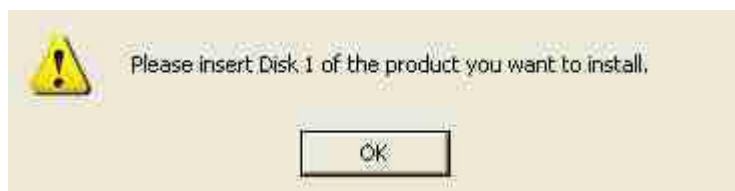




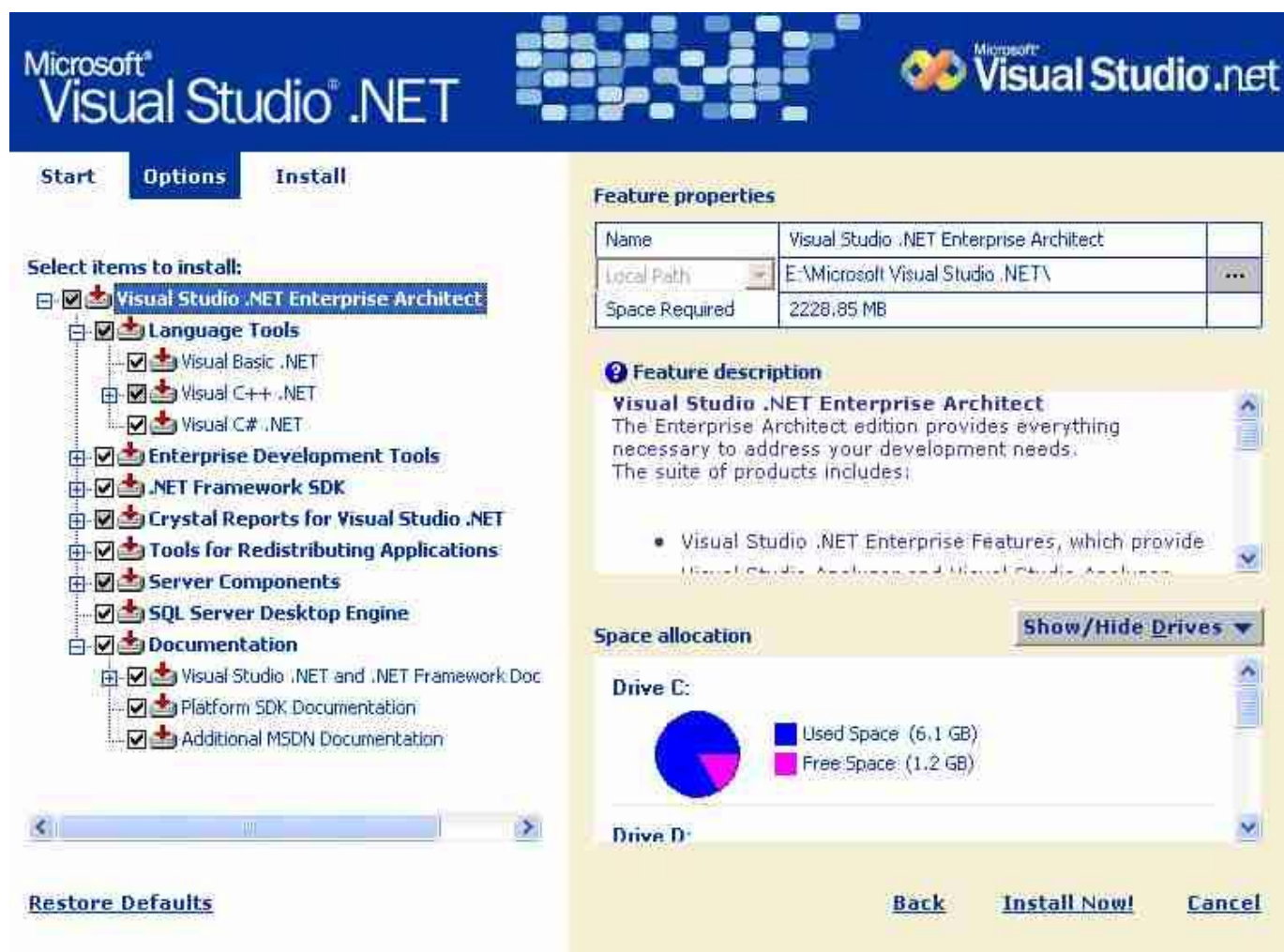
خود برنامه بصورت خودکار مولفه‌های مورد نیاز را تشخیص داده و تنها آن مولفه‌هایی را نصب می‌کند که بر روی سیستم شما وجود نداشته باشند. پس از ظاهر شدن صفحه فوق، بر روی **Install Now!** کلیک کنید و مرحله نصب این مولفه‌ها آغاز خواهد شد. پس از اتمام عمل نصب مولفه‌های مورد نظر، نوبت به نصب خود محیط **Visual Studio.Net** می‌رسد. پس از اتمام نصب مولفه‌های مورد نظر صفحه‌ای همانند شکل زیر ظاهر می‌گردد.



بر روی گزینه دوم، یعنی **Visual Studio.Net**، کلیک نمایید تا نصب برنامه آغاز گردد. البته توجه داشته باشید، آن دسته از دوستانی که دارای یک درایو **CD_ROM** هستند، قبل از انتخاب این گزینه **CD** شماره یک را در درایو خود قرار دهند، در غیر اینصورت با پیغام خطایی همانند شکل زیر روبرو خواهند شد.



پس از انتخاب نصب محیط **VS.Net**، صفحه باز شده و از شما شماره سریال برنامه و مشخصات فردی را درخواست می‌نماید. پس از وارد کردن این اطلاعات، بر روی گزینه **Continue** کلیک کنید. در این مرحله صفحه اصلی مربوط به انتخاب گزینه‌ها و ابزارهای مختلف جهت نصب محیط **VS.Net** ظاهر می‌گردد.



در این صفحه، قسمت‌های مختلفی مشاهده می‌شود که به توضیح هر یک از آنها خواهیم پرداخت. در قسمت چپ، امکان انتخاب نصب زبانهای مختلف برنامه‌سازی نظیر **Visual C++.Net**، **Visual Basic.Net** و **Visual C#.Net**

وجود دارد. در صورتیکه تنها می‌خواهید **C#** بر روی سیستم شما نصب گردد، تیک کنار زبانهای برنامه‌سازی دیگر را بردارید. ابزارهای متنوع دیگری نیز از قبیل **Crystal Report** و **Server Components** نیز در اینجا وجود دارند که در صورت تمایل می‌توانید آنها را انتخاب کنید. **Crystal Report** نرم‌افزاری است که با استفاده از آن به سادگی می‌توانید در برنامه‌های خود گزارشگیری کرده و فرمهای مختلفی از گزارشات را ایجاد نمایید. توصیه می‌کنم در صورتیکه تنها می‌خواهید **Visual C#.Net** بر روی سیستم شما نصب گردد، سایر ابزارها و امکانات مربوطه را نصب نمایید. در آخرین قسمت، گزینه‌ای تحت عنوان **Documentation** قرار دارد که همان **MSDN** و فایل‌های **Help** مربوط به محیط **VS.Net** و زبان‌های برنامه‌سازی مرتبط با آن است. در صورت تمایل می‌توانید این گزینه‌ها را نیز برای نصب بر روی سیستم خود انتخاب نمایید. توجه کنید که انتخاب این گزینه‌های، حدود 1,3 گیگابایت از فضای هارد دیسک شما را اشغال خواهد کرد. پس در صورتیکه نیازی به آنها ندارید از نصب آنها خودداری نمایید.

لیست مولفه‌ها و ابزارهای قابل نصب بشرح زیر می‌باشد :

1- Language Tools : در این شاخه می‌توانید زبان یا زبانهای مورد نظر را انتخاب نمایید. همانطور که قبلاً نیز گفته شد این زبانها در این نسخه از **VS.Net** شامل **VC++.Net**، **VB.Net** و **VC#.Net** می‌باشند.

2- Enterprise Development Tools : شامل ابزارها و مولفه‌های کمکی جانبی که برای محیط **VS.Net** در نظر گرفته شده‌اند. این شاخه خود شامل زیر شاخه‌های زیر است :

2-1 Visual Studio Enterprise Templates : Template های مختلف برای محیط **VS.Net**

2-2 Visual Studio SDK : شامل ابزارهای مختلفی جهت اضافه کردن ویزاها و ابزارهای جدید و شخصی به محیط **VS.Net**. با استفاده از ابزارهای این شاخه می‌توان به عنوان مثال فایل‌های **Help** برای برنامه خود ایجاد نمایید و آنرا به محیط برنامه خود بیفزایید.

2-3 Enterprise Samples : مثالهای نمونه‌ای که توسط **VS.Net** تولید شده‌اند.

2-4 Application Center Test : این ابزار بر روی ویندوزهای 2000، XP و **Advance Server 2003** و یا ویندوزهای مشابه نصب شده و امکان کنترل و جمع‌آوری اطلاعات مربوط به کارآیی و عملکرد برنامه‌های تحت وب را به شما می‌دهد. با استفاده از این ابزار امکان شبیه‌سازی برنامه تحت وب شما فراهم می‌شود و می‌توانید برنامه‌های تحت وب خود را قبل از ارسال بر روی وب، بر روی کامپیوتر خود تحت شرایط مختلف تست نمایید.

2-5 Visual Studio Analyzer : با استفاده از این ابزار، امکان آنالیز کارآیی برنامه، پیاده کردن خطاها، و درک ساختار برنامه‌های توزیع شده (**Distributed**) برای شما فراهم می‌گردد. با استفاده از این ابزار می‌توانید اطاعات مربوط به تمام رخدادهای برنامه را جمع‌آوری نمایید. بطور ویژه‌ای مشاهده کنید که برنامه توزیع شده شما چگونه عمل می‌کند و برنامه جهت جمع‌آوری رخدادهای برنامه تان ایجاد نمایید.

3- .Net Framework SDK : مرجع کاملی برای استفاده از **.Net Framework**. که شامل مثالهای عملی نیز می‌باشد.

4- **Crystal Report For Visual Studio.Net** : با استفاده از این برنامه بسیار مفید می‌توانید فرمها و ابزارها متنوعی جهت ایجاد گزارش از برنامه‌هایتان ایجاد نمایید.

5- **Tools For Redistributing Applications** : این شاخه حاوی زیر شاخه‌های زیر است:

5-1- **Graphic Library** : مجموعه‌ای شامل فایل‌های تصویری، آیکون و **cursor** جهت توسعه و ایجاد محیط‌های گرافیکی و GUI.

5-2- **Redistributable Merge Modules** : ماژول‌های لازم جهت نصب صحیح ابزارهای این قسمت.

6- **Server Components** : مولفه‌های لازم جهت تبدیل کامپیور شما به **Server** و یا **Client**. برای مثال این مولفه‌ها برای تولید و توسعه نرم‌افزارهای تحت وب و سرویس‌های وب بر روی کامپیوتر شما لازم هستند.

7- **SQL Server Desktop Engine** : ابزاری از خانواده **SQL** که می‌توانید بوسیله آن نرم‌افزارهای مخصوص پایگاه داده ایجاد نمایید.

8- **Documentations** : شامل مستندات و فایل‌های **Help** و همچنین **MSDN** مربوط به **VS.Net**.

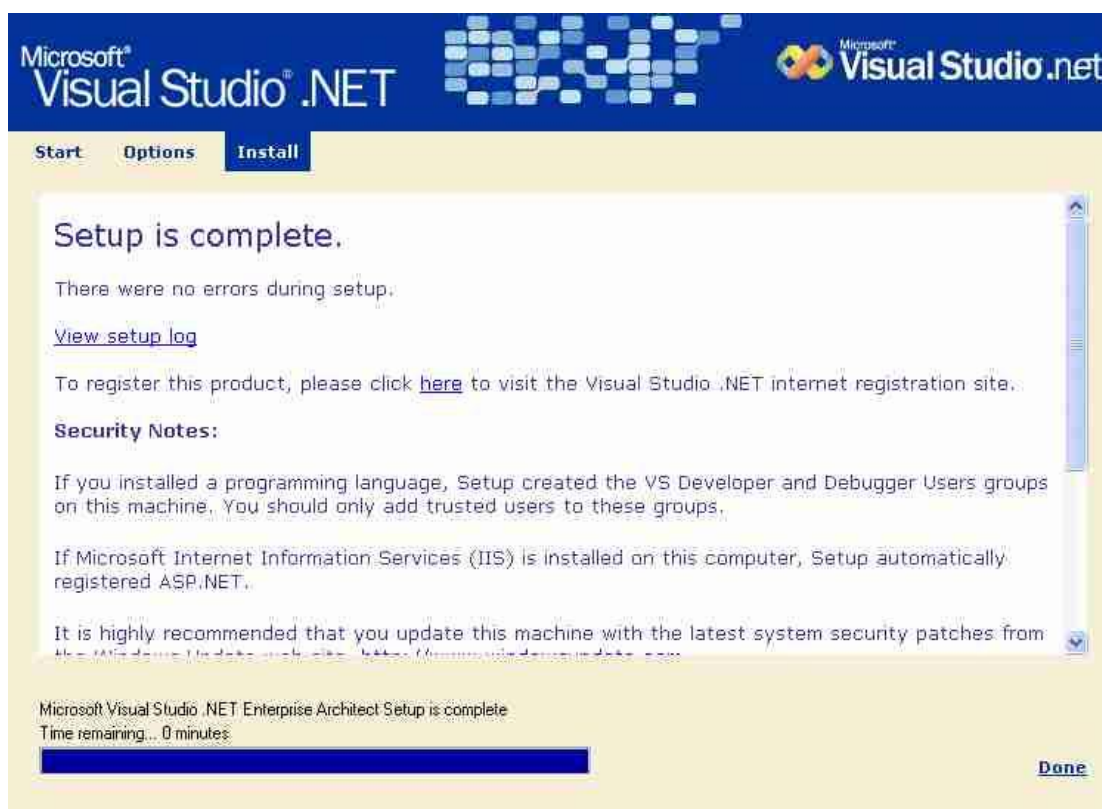
در سمت راست صفحه نیز می‌توانید محل نصب برنامه را تعیین کنید. فضای مورد نیاز برای نصب گزینه‌های انتخاب شده نیز، در این قسمت به شما نشان داده می‌شود. در صورتیکه مسیری را که برای نصب انتخاب نموده‌اید، دارای فضای کافی برای نصب تمامی مولفه‌های انتخاب شده نباشد، بر روی درایو هارد دیسک شما علامت ضرب در قرمزی دیده می‌شود که نشان دهنده آنست که بر روی درایو مورد نظر، فضای کافی وجود ندارد. پس از انجام این مراحل گزینه **Install Now!** را انتخاب کنید تا نصب برنامه آغاز گردد.

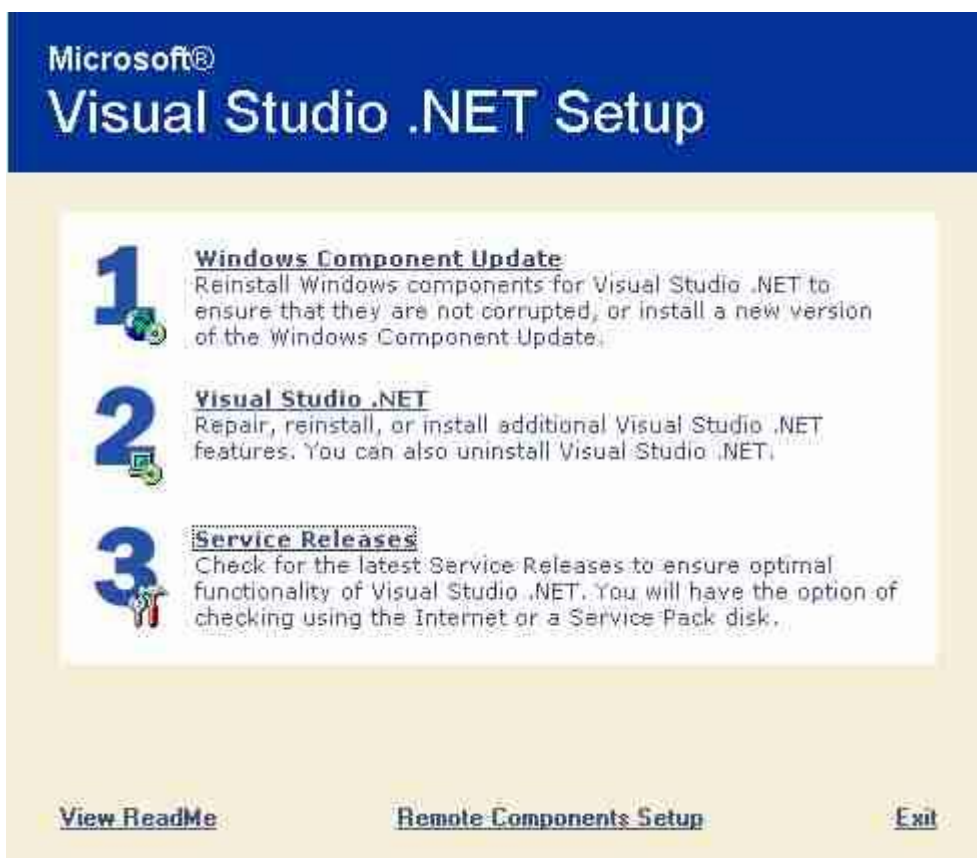


در حین عمل نصب برنامه، پیامهایی منبئ بر تعویض CD موجود در درایو نشان داده خواهند شد که می‌بایست در این هنگام، CD مورد نظر را درون درایو خود قرار دهید.



پس از اتمام عمل نصب برنامه، صفحه زیر ظاهر می‌شود و پس از انتخاب گزینه **Done** صفحه دیگری ظاهر می‌شود که در آن گزینه 3 یعنی **Service Release** بصورت فعال دیده می‌شود. این گزینه مربوط به مشاهده آپدیت‌ها و تغییرات جدیدی است که برای محیط **VS.Net** وجود دارد و شما را به سایت **Microsoft** هدایت خواهد کرد. با انتخاب گزینه **Exit** عمل نصب به پایان خواهد رسید.





پس از اتمام مراحل نصب، می‌توانید **Visio** و **Source Safe** را نیز بر روی کامپیوتر خود نصب نمایید. نصب این دو برنامه بسیار ساده و شبیه به برنامه‌های عادی است. توجه نمایید که از **Visio** موجود در پک **Office** نمی‌توان برای محیط **VS.Net** استفاده کرد، از اینرو آندسته از افرادی که می‌خواهند تحت **.Net** با **Visio** کار کنند، حتماً باید نسخه مخصوص **VS.Net** را تهیه نمایند.

کامپایل یک برنامه C#

اجرای برنامه‌های **C#** از طریق کامپایلر **C#** دو نوع اطلاعات مهم ایجاد می‌کند: **کد** و **metadata**. در مطلب زیر درباره این دو نوع اطلاعات بحث خواهیم نمود.

زبان سطح میانی میکروسافت **MSIL**

کدی که توسط کامپایلر **C#** تولید می‌شود به زبان خاصی است که به آن، زبان سطح میانی میکروسافت یا **MSIL** می‌گویند. این زبان مجموعه‌ای از دستورالعمل‌ها است که نحوه اجرای برنامه شما را معین می‌کند. این کد حاوی دستورالعمل‌هایی جهت مقداردهی متغیرها، فراخوانی اشیاء، متدها و کنترل خطا می‌باشد. باید به این نکته نیز اشاره کرد که زبان **C#** تنها زبانی نیست که کد آن به **MSIL** تبدیل می‌شود، بلکه کد تمامی زبانهای تحت **.Net** به زبان **MSIL** تبدیل می‌شوند، یعنی به محض کامپایل، کدهای نوشته شده توسط شما به کد زبان **MSIL** تبدیل می‌شود. به دلیل اینکه

کد تمامی زبانهای تحت **Net**. به دستورالعملهای مشابهی در **MSIL** تبدیل می‌شوند و به دلیل اینکه تمامی این زبانها دارای محیط اجرایی یکسانی هستند، از اینرو این زبانها به راحتی می‌توانند با یکدیگر ارتباط برقرار نمایند و از کدهای یکدیگر استفاده کنند و بدون مشکل با یکدیگر کار کنند.

یکی از مزایای مهم **MSIL** اینست که دستورالعملهای آن مخصوص پردازنده‌ای (**CPU**) خاص طراحی نشده است. به بیان دیگر **MSIL** هیچ چیز در مورد **CPU** ماشینی که بر روی آن اجرا می‌شود نمی‌داند و **CPU** ماشین نیز چیزی در مورد **MSIL** نمی‌داند. حال سوالی که در اینجا مطرح می‌شود آنست که پس **CPU** چگونه کدهای **MSIL** را اجرا می‌کند؟ باید گفت که کدهای **MSIL** بر روی هر ماشینی، به دستورالعملهای خاص آن ماشین تبدیل می‌گردند. این عمل، که تنها در اولین اجرای برنامه بر روی ماشینی خاص اتفاق می‌افتد را با نام "کامپایل آنی" یا **JIT** می‌گوئیم. وظیفه کامپایلر **JIT** نیز تبدیل کدهای **MSIL** به دستورالعملهای خاص ماشینی است که برنامه بر روی آن اجرا می‌شود.

همانطور که ملاحظه می‌کنید این مرحله بسیار جالب است. اما شاید این سوال در ذهن شما مطرح شود که، زمانیکه کامپایلر می‌تواند کدی منطبق با دستورالعملهای پردازنده‌ای که روی آن اجرا می‌شود تولید نماید، پس چه لزومی به تبدیل این کدها به زبان **MSIL** وجود دارد؟ این عمل دقیقاً عملی است که در گذشته و توسط کامپایلرهای قدیمی انجام می‌شد، اما دلایلی برای تبدیل کدها به زبان **MSIL** وجود دارد. اولین و مهمترین دلیل آنست که کدهای کامپایل شده به زبان **MSIL** را می‌توان به سادگی به سخت‌افزار دیگری نیز منتقل نمود. برای مثال تصور کنید برنامه‌ای را بر روی کامپیوتر شخصی خود به زبان **C#** نوشته و می‌خواهید آنرا به کامپیوتر جیبی خود منتقل نمایید. همانطور که می‌دانید این دو نوع کامپیوتر دارای پردازنده‌هایی متفاوت هستند و دستورالعملهای هر یک از این پردازنده‌ها نیز با یکدیگر متفاوت است. از اینرو برنامه شما فقط بر روی کامپیوتر شخصی قابل اجرا خواهد بود و نمی‌توانید آنرا بر روی سخت‌افزار دیگری منتقل کنید. در این شرایط شما به دو کامپایلر نیاز خواهید داشت: یک کامپایلر که برنامه **C#** شما را برای **CPU** کامپیوتر شخصی کامپایل کند و کامپایلر دیگری که این برنامه را برای کامپیوتر جیبی کامپایل نماید. همچنین در این حالت باید برنامه خود را دوبار کامپایل نمایید. اما با استفاده از **MSIL** تنها یکبار برنامه خود را کامپایل می‌کنید. در این حالت با نصب **.Net Framework** و **JIT** بر روی کامپیوتر شخصی خود، برنامه‌تان اجرا می‌شود و با نصب آن بر روی کامپیوتر شخصی، برنامه شما اجرا می‌شود. حال شما کد **MSIL** ای دارید که بر روی هر ماشینی که دارای کامپایلر **JIT** مربوط به **.Net Framework** است، اجرا می‌شود.

یکی دیگر از مزایای استفاده از زبان **MSIL** در آنست که کدهای تولید شده در این مرحله به سادگی در پروسه تایید کد (**Code Verification**) قابل خواندن هستند. پروسه تایید کد، پروسه‌ای است که در آن **JIT** کد تولید شده را بررسی می‌کند تا فاقد هرگونه خطا و مشکل باشد. از جمله بررسی‌هایی که در این مرحله انجام می‌شود عبارتند از: بررسی دسترسی صحیح به حافظه، استفاده از متغیر صحیح، تبدیل صحیح انواع مختلف به یکدیگر و این بررسی‌ها باعث می‌شوند تا برنامه‌ها به‌نگام اجرا دچار مشکل نشده و با موفقیت اجرا شوند. تولید کدهای مختص به یک نوع پردازنده، هرچند سرعت اجرا را با بالا می‌برد اما کدی تولید می‌کند که بررسی آن بسیار دشوار است. تبدیل کدهای نوشته

شده به زبان **C#** به کدی مختص پردازنده‌ای خاص باعث ایجاد کدی می‌گردد که در مرحله تایید کد غیر قابل تشخیص است و یا به سختی قابل بررسی است.

Metadata

پروژه کامپایل، علاوه بر کد اصلی برنامه، حاوی خروجی دیگری با عنوان **Metadata** می‌باشد که بخش بسیار مهمی در به اشتراک‌گذاری کدهای **.Net** می‌باشد. چه بخواهید تا برنامه‌ای مستقل با **C#** تولید کنید و چه بخواهید کلاسی تولید کنید که در برنامه‌ای دیگر مورد استفاده قرار گیرد، ممکن است بخواهید تا از کدهایی که قبلاً کامپایل شده‌اند استفاده نمایید. این کد یا توسط مایکروسافت بعنوان بخشی از **.Net Framework** پشتیبانی می‌شوند و یا توسط شخصی دیگر منطبق با **.Net** تولید شده‌اند. زمانیکه بخواهید از این کد استفاده کنید، کامپایلر **C#** شما باید اطلاعاتی درباره متغیرها و نوع متغیرهای بکار رفته در این کد در اختیار داشته باشد تا بتواند خود را با آن منطبق نماید.

Metadata را می‌توان بعنوان فهرست اطلاعات موجود در کدی که قبلاً کامپایل شده در نظر گرفت. **C#** فایل **Metadata** را در هنگام تولید کد **MSIL** تولید می‌کند و در آن اطلاعات مفیدی را قرار می‌دهد. این فایل حاوی اطلاعاتی در مورد کلیه کلاسهای کد تولید شده و ساختار آنها می‌باشد. همچنین در **Metadata** تمامی متدها و متغیرهای بکار رفته در کلاس بطور کامل توصیف شده و برای برنامه‌های دیگر قابل خواندن است. بعنوان مثال یک برنامه می‌تواند با استفاده از **Metadata** موجود، لیست کلیه متدهای موجود را استخراج کرده و مورد استفاده قرار گیرد. تمامی اطلاعات لازم جهت توصیف و بررسی یک کلاس در **Metadata** قرار می‌گیرد.

اسمبلی‌ها (Assemblies)

در اکثر موارد، شما از زبان **C#** بعنوان زبانی برای تولید برنامه‌های کاربردی استفاده می‌کنید. این برنامه‌های کاربردی بصورت فایل‌هایی اجرایی در می‌آیند که دارای پسوند **.EXE** هستند. ویندوز همواره فایل‌های اجرایی را با پسوند **.EXE** می‌شناسد و **C#** نیز تولید اینگونه فایل‌ها را کاملاً پشتیبانی می‌کند.

اما زمانهایی هستند که نیازی به تولید کل یک برنامه نیست و تنها لازم است تا کلاس و یا ابزاری برای یک برنامه جامع‌تر تولید شود. برای مثال فرض کنید می‌خواهید ابزاری با زبان **C#** تولید کنید و آنرا به گروهی بدهید که با **VB.Net** کار می‌کنند. در چنین شرایطی بجای تولید فایل اجرایی **.EXE**، فایل اسمبلی تولید می‌شود.

اسمبلی بسته‌ای حاوی کد و **Metadata** است. هنگامیکه مجموعه‌ای از کلاسها را در یک اسمبلی جمع می‌کنید، تمامی این کلاسها بعنوان یک واحد در نظر گرفته می‌شوند و دارای سطوح یکسانی هستند. می‌توان به اسمبلی بعنوان یک **DLL** منطقی (**Logical DLL**) نگریست. در **.Net** بحث اسمبلی بسیار شبیه به **Microsoft Transaction Server** و **COM+** است.

بطور کلی دو نوع اسمبلی وجود دارد: اسمبلی‌های خصوصی و اسمبلی‌های عمومی. در هنگام تولید اسمبلی نیازی به مشخص کردن نوع اسمبلی نیست. اسمبلی‌های خصوصی تنها برای یک برنامه قابل دسترس هستند. در این حالت اسمبلی شما بصورت یک **DLL** ایجاد می‌شود و در دایرکتوری مشابه با برنامه‌ای که از آن استفاده می‌کند، نصب می‌گردد. با تولید اسمبلی خصوصی (**Private**) تنهای برنامه‌ای که می‌تواند از اسمبلی شما استفاده کند، برنامه اجرایی است که در دایرکتوری مشابه با اسمبلی شما قرار داشته باشد.

اما اگر بخواهید برنامه‌های متعددی از کد شما استفاده کنند، از اسمبلی عمومی (**Global**) استفاده می‌کنید. اسمبلی‌های عمومی توسط کلیه برنامه‌های **.Net** موجود بر روی سیستم قابل استفاده هستند. مایکروسافت این اسمبلی‌ها را بعنوان اسمبلی‌های **.Net** در نظر می‌گیرد و کلیه اسمبلی‌های **.Net** برای کلیه برنامه‌ها قابل استفاده هستند. **.Net Framework** شامل لیستی از اسمبلی‌های عمومی است که تحت عنوان **global assembly cache** شناخته می‌شوند و **.Net Framework SDK** شامل ابزارهایی جهت نصب و پاک کردن اسمبلی‌های مختلف از این دایرکتوری می‌باشد.

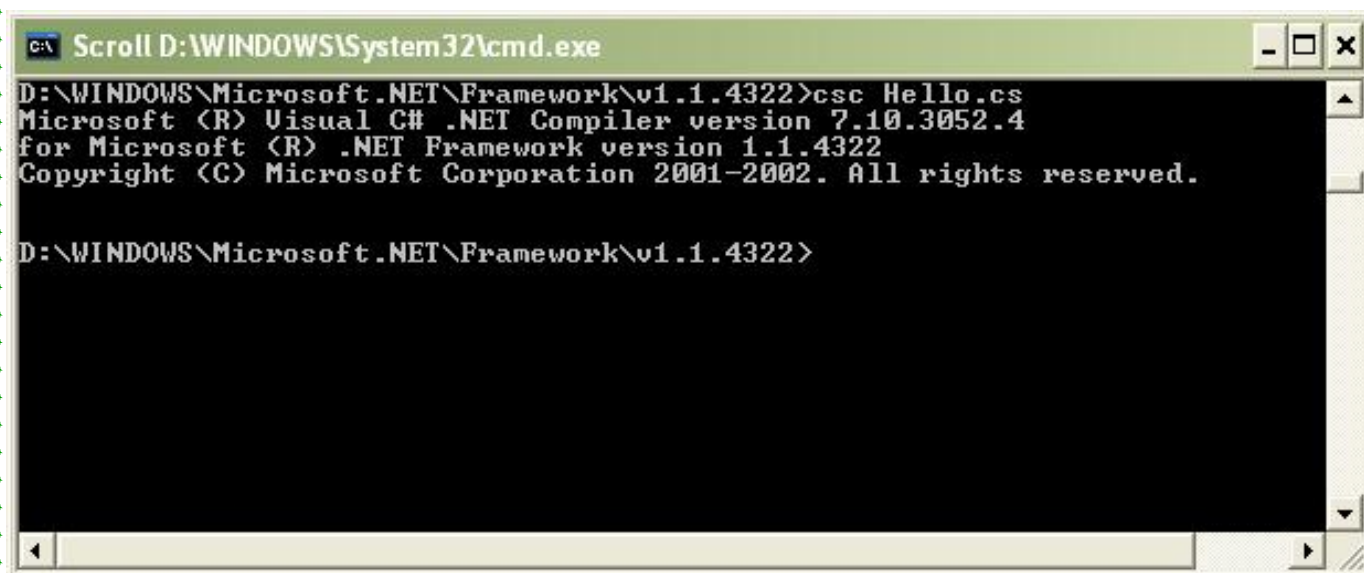
کامپایل برنامه در *Command Line*

پس از اینکه برنامه خود را نوشتید، نوبت به کامپایل آن می‌رسد. توجه کنید که برای کامپایل برنامه‌های **C#** حتماً به محیط ویژوال نیازی نیست. چون دوستان بسیاری به من ایمیل زدند و درخواست کرده‌اند تا نحوه کامپایل برنامه‌ها را در **Command Line** را توضیح دهم، در اینجا به توضیح درباره آن پرداختم. توجه کنید که شما برای شروع کار حتی به محیطی مثل **VS.Net** و یا **C#Builder** هم نیازی ندارید. تنها کافیست تا کد برنامه خود را در یک ویرایشگر متن مانند **Notepad** تایپ کنید و سپس از طریق **Command Line** به اجرای آن پردازید. تنها نکته‌ای که باید به آن توجه نمایید آنست که در هنگام ذخیره کردن کدی که در **Notepad** نوشته اید، باید به انتهای نام فایل خود پسوند **.cs** را اضافه نمایید.

حال نوبت به کامپایل کد نوشته شده می‌رسد. کامپایلر **Command Line** بطور پیش فرض در دایرکتوری زیر وجود دارد: **c:\windows\Microsoft.Net\Framework\v.xxxx** تحت عنوان **csc.exe** قرار دارد که **xxxx** شماره ورژن **.Net Framework** نصب شده بر روی سیستم شماست. فرض کنید می‌خواهیم برنامه‌ای با نام **Hello.cs** را که در محیط **Notepad** آنرا نوشته‌ایم را کامپایل کنیم. وارد **Command Line** شده و عبارت **csc Hello.cs** را تایپ کنید. پس از این دستو در صورتیکه کد شما دارای خطا نباشد، در همان دایرکتوری موجود فایل اجرایی تحت عنوان **Hello.exe** تولید خواهد شد که همان برنامه کامپایل شده شماست.

نکته مهمی که باید به آن توجه کنید، وجود کامپایلر در مسیری است که فایل کد مورد نظر شما قرار دارد. برای مثال در صورتیکه **csc.exe** را به مسیر ویندوز اضافه نکرده باشید، برای اجرای مثال قبل باید فایل نوشته شده را در مسیر

csc.exe قرار دهید و سپس از طریق **Command Line** آنرا اجرا نمایید. برای مثال اجرای این برنامه بر روی کامپیوتر خود من بشکل زیر است. (علت اینکه گفته‌ام بر روی کامپیوتر خود من، بخاطر متفاوت بودن مسیرهای نصب برنامه بر روی کامپیوتر های مختلف است.)



```
C:\> Scroll D:\WINDOWS\System32\cmd.exe
D:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>csc Hello.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

D:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>
```

همانطور که مشاهده می‌کنید، پس از اجرای دستور گفته شده، در صورتیکه برنامه بدون خطا باشد، پیغام خطایی ظاهر نشده و فایل اجرایی برنامه تولید می‌شود.

کدی که من برای **Hello.cs** در نظر گرفته بودم بصورت زیر است :

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World!");
        System.Console.ReadLine();
    }
}
```

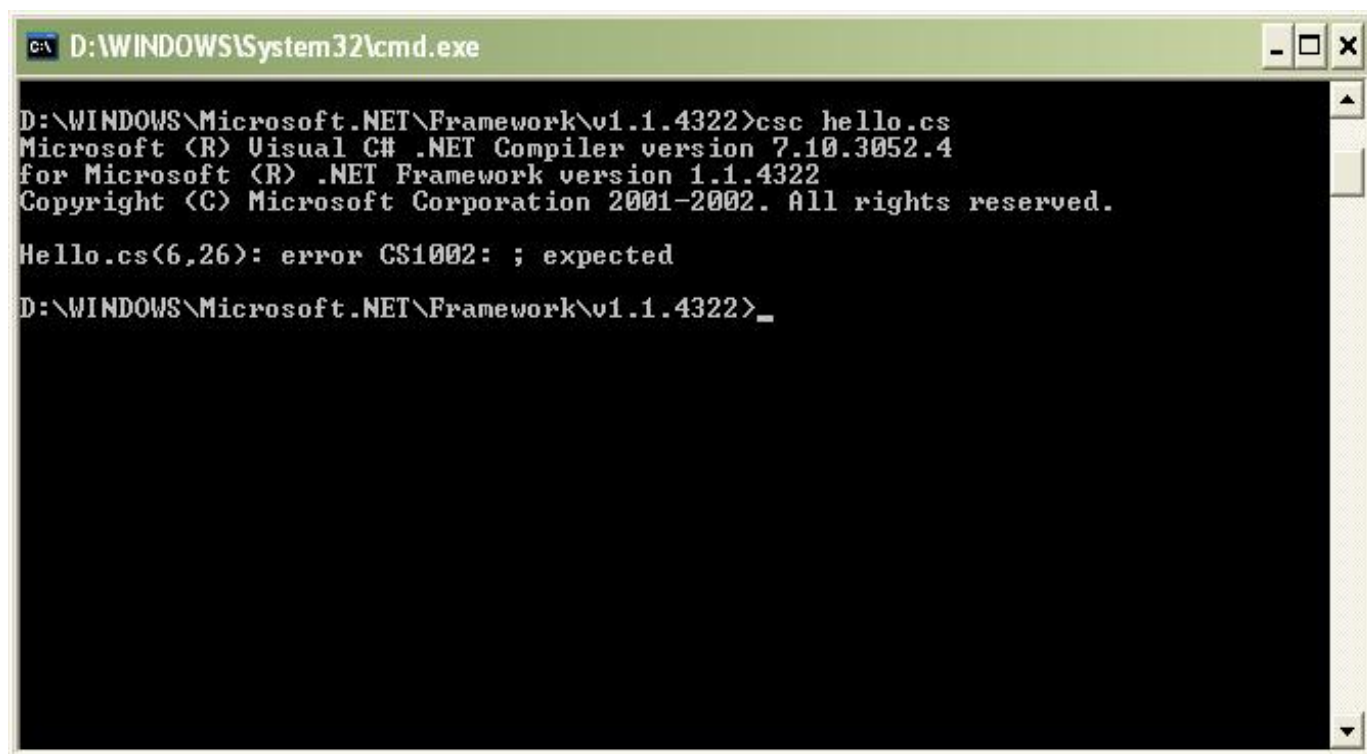
با اجرای کد فوق فایل اجرایی آن ایجاد شده و قابل استفاده می‌گردد. در صورتیکه کد شما دارای خطا باشد، پس از کامپایل کد از طریق **Command Line**، خطی از کد که خطا در آن وجود دارد با نوع خطای آن برای شما نمایش داده می‌شود.

```

class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World!");
        System.Console.ReadLine()
    }
}

```

برای مثال در کد فوق، در خط آخر سمی کالون ";" را حذف کرده‌ام تا خطایی اتفاق افتد. خروجی کامپایلر بصورت شکل زیر در خواهد آمد.



The screenshot shows a Windows Command Prompt window titled "D:\WINDOWS\System32\cmd.exe". The command prompt displays the following text:

```

D:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>csc hello.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

Hello.cs(6,26): error CS1002: ; expected
D:\WINDOWS\Microsoft.NET\Framework\v1.1.4322>_

```

همانطور که در شکل بالا مشاهده می‌کنید، پس از اجرای کامپایلر C#، خطایی ظاهر می‌شود که در آن نشان می‌دهد که برنامه در سطر 6 و ستون 26 دارای خطا است که این خطا عدم وجود ";" است.

کار با کامپایلر **Command Line** اندکی دشوار بوده و نیاز به تمرین و اندکی تامل دارد

درس اول : آغاز کار با C#

در این درس با ارائه چند برنامه و مثال ساده به طرز کار زبان C# می‌پردازیم. اهداف این درس عبارتند از :

- فهم ساختار پایه‌ای یک برنامه C#
- آشنایی با Namespace
- آشنایی با کلاس (Class)
- آشنایی با عملکرد متد (Main)
- آشنایی با ورودی/خروجی یا I/O

لیست 1-1، یک برنامه ساده با عنوان **Welcome** در زبان C#

// **Namespace** اعلان

using System;

// کلاس آغازین برنامه

class WelcomeCSS

{

// آغاز کار اجرای برنامه

public static void Main()

{

// نوشتن متن در خروجی

Console.WriteLine("Welcome to the C# Persian Tutorial!");

}

}

برنامه لیست 1-1 دارای 4 پارامتر اصلی است، اعلان **Namespace**، کلاس، **Main()** متد و یک دستور زبان C#. در همین جا باید به یک نکته اشاره کنم، برای زبان C# همانند بیشتر زبانهای برنامه‌سازی دو نوع کامپایلر وجود دارد. یک نوع کامپایلر که به کامپایلر **Command Line** معروف است و نوع دیگر کامپایلر **Visual** است. کامپایلرهای **Command Line** محیطی شبیه به محیط DOS دارند و با دادن یک سری دستورات به اجرا در می‌آیند. کامپایلرهای **Visual** محیطی همانند ویندوز دارند که با دارا بودن محیط گرافیکی و ابزارهای خاص، برنامه‌نویس را در امر برنامه‌سازی کمک می‌کنند. از نمونه‌های هر یک از کامپایلرها، می‌توان به **Microsoft C# Command Line** و **Compiler** که یک کامپایلر **Command Line** و **Microsoft Visual C#** که یک کامپایلر **Visual** است، اشاره کرد. البته در حال حاضر بیشتر از کامپایلرهای ویژوال استفاده می‌شود.

من سعی می‌کنم در آینده به توضیح محیط **Visual C#** و **Visual Studio.Net** پردازم. اما فعلاً برای اجرای برنامه‌ها می‌توانید از **Visual Studio.Net** استفاده کنید. پس از نصب آن، وارد محیط **C#** شده و در قسمت انتخاب برنامه جدید گزینه **Console** را جهت اجرای برنامه‌ها انتخاب نمایید.

برای این درس، فعلاً به توضیحات بیشتر درباره محیط ویژوال نمی‌پردازم اما در آینده به توضیح کامل محیط **Visual Studio.Net** خواهم پرداخت.

برای اجرای کد بالا در صورتیکه از محیط ویژوال استفاده می‌کنید باید بر روی دکمه **Run** کلیک کنید و در صورتیکه کامپایلر **Command Line** دارید با دستور زیر می‌توانید برنامه را اجرا کنید : **csc Welcome.cs**

پس از اجرای برنامه، کامپایلر برای شما یک فایل قابل اجرا (**Executable**) تحت نام **Welcome.exe** تولید می‌کند.

نکته : در صورتیکه از **Visual Studio.Net(VS.Net)** استفاده کنید، پس از اجرای برنامه، یک صفحه برای نمایش خروجی به سرعت باز شده و بسته می‌شود و شما قادر به دیدن خروجی نخواهید بود. برای اینکه بتوانید خروجی برنامه را ببینید، در انتهای برنامه دستور زیر را وارد نمایید :

Console.ReadLine();

استفاده از این دستور باعث می‌شود تا برنامه منتظر دریافت یک ورودی از کاربر بماند، که در این حالت شما می‌توانید خروجی برنامه خود را دیده و سپس با زدن کلید **Enter** برنامه را خاتمه دهید.

نکته دیگری که در مورد زبان برنامه‌نویسی **C#** باید مورد توجه قرار دهید اینست که این زبان **Case Sensitive** است، بدین معنا که به حروف کوچک و بزرگ حساس است یعنی برای مثال **ReadLine** با **readLine** متفاوت است به طوریکه اولی جزو دستورات زبان **C#** و دومی به عنوان یک نام برای متغیر یا یک تابع که از طرف کاربر تعریف شده است در نظر گرفته می‌شود.

اعلان **Namespace** به سیستم اعلان می‌نماید که شما از توابع کتابخانه‌ای **System** جهت اجرای برنامه‌ها خود استفاده می‌نمایید. دستوراتی مانند **WriteLine** و **ReadLine** جزو توابع کتابخانه‌ای **System** می‌باشند. اغلب دستورات و توابع مهم و کلیدی استفاده از کنسول ورودی/خروجی در این کتابخانه می‌باشد. در صورتیکه در ابتدای برنامه از **using System** استفاده نکنید، باید در ابتدای هر یک از دستورات برنامه که مربوط این کتابخانه است، از کلمه **System** استفاده نمایید. بعنوان مثال در صورت عدم استفاده از **using System** باید از دستور **System.Console.WriteLine()** به جای **Console.WriteLine()** استفاده نمایید.

تعریف کلاس، **Class Welcome CSS**، شامل تعریف داده‌ها (متغیرها) و متدها جهت اجرای برنامه است. یک کلاس، جزو محدود عناصر زبان **C#** است که بوسیله آن می‌توان به ایجاد یک شی (**Object**) از قبیل واسط‌ها (**Interfaces**) و ساختارها (**Structures**)، پرداخت. توضیحات بیشتر در این زمینه در درس‌های آینده ذکر می‌شوند. در این برنامه کلاس هیچ داده و متغیری ندارد و تنها شامل یک متد است. این متد، رفتار (**Behavior**) این کلاس را مشخص می‌کند.

متد درون این کلاس بیان می‌کند که این کلاس چه کاری را پس از اجرا شدن انجام خواهد داد. کلمه کلیدی **Main()** نام متد این کلاس نیز می‌باشد جزو کلمات رزرو شده زبان **C#** است که مشخص می‌کند برنامه از کجا باید آغاز به کار نماید. وجود متد **Main()** در تمامی برنامه‌های اجرایی ضروری است. در صورتیکه یک برنامه حاوی متد **Main()** نباشد بعنوان توابع سیستمی همانند **dll** های ویندوز در نظر گرفته می‌شود.

قبل از کلمه **Main()** کلمه دیگری با عنوان **static** آورده شده است. این کلمه در اصطلاح **Modifier** می‌گویند. استفاده از **static** برای متد **Main()** بیان می‌دارد که این متد تنها در در همین کلاس قابل اجراست و هیچ نمونه‌ای **(Instance)** دیگری از روی آن قابل اجرا نمی‌باشد. استفاده از **static** برای متد **Main()** الزامی است زیرا در ابتدای آغاز برنامه هیچ نمونه‌ای از هیچ کلاس و شی‌ای موجود نمی‌باشد و تنها متد **Main()** است که اجرا می‌شود. (در صورتیکه با برخی اصطلاحات این متن از قبیل کلاس، شی، متد و نمونه آشنایی ندارید، این به دلیل آنست که این مفاهیم جزو مفاهیم اولیه برنامه‌نویسی شی گرا **(OOP)** هستند. سعی می‌کنم در درس‌های آینده به توضیح این مفاهیم نیز بپردازم، ولی فعلاً در همین حد کافی می‌باشد).

هر متد باید دارای یک مقدار بازگشتی باشد، یعنی باید مقداری را به سیستم بازگرداند، در این مثال نوع بازگشتی **void** تعریف شده است که نشان دهنده آنست که این متد هیچ مقداری را باز نمی‌گرداند یا به عبارت بهتر خروجی ندارد. همچنین هر متد می‌تواند دارای پارامترهایی نیز باشد که لیست پارامترهای آن در داخل پرانتزهای جلوی آن قرار می‌گیرد. برای سادگی کار در این برنامه متد ما دارای هیچ پارامتری نیست ولی در ادامه همین درس به معرفی پارامترها نیز می‌پردازم.

متد **Main()** رفتار و عمل خود را بوسیله **Console.WriteLine(...)** مشخص می‌نماید. **Console** کلاسی در **System** است و **WriteLine()** متدی در کلاس **Console**. در زبان **C#** از اپراتور "." (نقطه dot) جهت جداسازی زیرروتین‌ها و زیرقسمتها استفاده می‌کنیم. همانطور که ملاحظه می‌کنید چون **WriteLine()** یک متد درون کلاس **Console** است به همین جهت از "." جهت جداسازی آن استفاده کرده‌ایم.

در زبان **C#**، برای قرار دادن توضیحات در کد برنامه از // استفاده می‌کنیم. بدین معنا که کامپایلر در هنگام اجرای برنامه توجهی به این توضیحات نمی‌کند و این توضیحات تنها به منظور بالا بردن خوانایی متن و جهت و کمک به فهم بهتر برنامه قرار می‌گیرند. استفاده از // تنها در مواردی کاربرد دارد که توضیحات شما بیش از یک خط نباشد. در صورت تمایل برای استفاده از توضیحات چند خطی باید در ابتدای شروع توضیحات از /* و در انتها آن از */ استفاده نمایید. در این حالت تمامی مطالبی که بین /* */ قرار می‌گیرند به عنوان توضیحات **(Comments)** در نظر گرفته می‌شوند.

تمامی دستورات **(Statements)** با ";"، سمی کولون، پایان می‌یابند. کلاس‌ها و متدها با { آغاز شده و با } خاتمه می‌یابند. تمامی دستورات بین { } یک بلوک را می‌سازند.

بسیاری از برنامه‌ها از کاربر ورودی دریافت می‌کنند. انواع گوناگونی از این ورودی‌ها می‌توانند به عنوان پارامتری برای متد **Main()** در نظر گرفته شوند. لیست 1-2 برنامه‌ای را نشان می‌دهد نام کاربر را از ورودی دریافت کرده و آن را بر روی صفحه نمایش می‌دهد. این ورودی به صورت پارامتری برای متد **Main()** در نظر گرفته شده است.

لیست 1-2: برنامه‌ای که ورودی را از کاربر، بعنوان پارامتر **Main()**، دریافت می‌کند.

// Namespace اعلان

using System;

// کلاس آغازین برنامه

class NamedWelcome

{

// آغاز اجرا برنامه

public static void Main(string[] args)

{

// نمایش بر روی صفحه

Console.WriteLine("Hello, {0}!", args[0]);

Console.WriteLine("Welcome to the C# Persian Tutorial!");

}

}

توجه داشته باشید که این برنامه، ورودی را به صورت **Command-Line** دریافت می‌کند و در هنگام اجرای برنامه باید ورودی را در **Command-Line** وارد نمایید. در صورتیکه ورودی را وارد ننمایید برنامه دچار مشکل شده و متوقف خواهد شد.

همان طور که در لیست 1-2 مشاهده می‌نمایید، پارامتر متد **Main()** با عنوان **args** مشخص شده است. با استفاده از این نام در داخل متد می‌توان آن استفاده نمود. نوع این پارامتر از نوع آرایه‌ای از نوع رشته (**string[]**) در نظر گرفته شده است. انواع (**types**) و آرایه‌ها را در درس‌های بعدی بررسی می‌کنیم. فعلاً بدانید که آرایه رشته‌ای جهت نگهداری چندین کاراکتر مورد استفاده قرار می‌گیرد. [] مشخص کننده آرایه هستند که مانند یک لیست عمل می‌کند.

همانطور که ملاحظه می‌کنید در این برنامه دو دستور **Console.WriteLine(...)** وجود دارد که اولین دستور مقداری با دستور دوم متفاوت است. همانطور که مشاهده می‌کنید داخل دستور **Console.WriteLine(...)** عبارتی به شکل **{0}** وجود دارد. این آرگومان، نشان می‌دهد که به جای آن چه مقداری باید نمایش داده شود که در این جا **args[0]** نشان داده می‌شود. عبارتی که داخل " " قرار دارد عیناً در خروجی نمایش داده می‌شود، به جای آرگومان **{0}**، مقداری که پس از " قرار دارد، قرار می‌گیرد. حال به آرگومان بعدی یعنی **args[0]** توجه کنید. مقدار صفر داخل [] نشان می‌دهد که کدام عنصر از آرایه مورد استفاده است. در **C#** اندیس آرایه از صفر شروع می‌شود به همین جهت برای دسترسی به اولین عنصر آرایه باید از اندیس صفر استفاده کنیم. (همانطور که قبلاً نیز گفتیم آرایه‌ها را در درس‌های آینده توضیح خواهیم داد، هدف از این درس تنها آشنایی با **C#** است!).

مجدداً به آرگومان **{0}** بازگردیم. این آرگومان نشان می‌دهد که یک مقدار باید در رشته خروجی قرار گیرد، این مقدار همان **args[0]** است. اگر شما این برنامه را از طریق **Command-Line** اجرا نمایید خروجی شبیه به زیر خواهید گرفت:

>Hello!, Meysam!

>Welcome to C# Persian Tutorial!

همان گونه که می بینید، پس از اجرای برنامه نام شما که از طریق **Command-Line** آنرا وارد نموده اید در خروجی ظاهر می شود. استفاده از آرگومان **{n}**، که در آن **n** یک مقدار عددی است، جهت فرمت دادن به متن خروجی است که بر روی صفحه به نمایش در می آید. مقدار **n** از صفر آغاز شده و به ترتیب افزایش می یابد. به مثال زیر توجه کنید :

```
Console.WriteLine("Hello! ,{0} ,{1}, {2}",args[0],args[1],args[2]);
```

این خط از برنامه سه مقدار **args[0],args[1],args[2]** را در خروجی به ترتیب نمایش می دهد. ملاحظه می نمایید که چون 3 مقدار را می خواهیم نمایش دهیم، سه بار از آرگومان **{n}** استفاده کرده ایم و هر بار یک واحد به مقدار قبلی افزوده ایم. در آینده بیشتر با این مفاهیم آشنا می شویم.

مطلبی که باید در مورد لیست 1-2 به آن توجه شود آنست که این برنامه تنها از طریق **Command-Line** قابل اجراست و در صورتیکه کاربر از این مطلب که برنامه باید دارای ورودی به صورت **Command-Line** باشد، بی اطلاع باشد و ورودی را در **Command-Line** وارد نکند، برنامه متوقف شده و اجرا نمی شود. پس برای رفع چنین مشکلی باید از روش بهتری جهت دریافت ورودی از کاربر استفاده کرد.

لیست 1-3 : یک برنامه که قابلیت محاوره با کاربر را دارد.

// Namespace اعلان

using System;

// کلاس آغازین برنامه

class InteractiveWelcome

{

// آغاز اجرای برنامه

public static void Main()

{

// متنی بر روی صفحه نمایش داده می شود

Console.Write("What is your name?: ");

// متنی نمایش داده شده و برنامه منتظر دریافت ورودی می ماند

Console.Write("Hello, {0}! ", Console.ReadLine());

Console.WriteLine("Welcome to the C# Persian Tutorial!");

}

}

همانطوریکه در این برنامه دیده می شود، متد **Main()** دارای پارامتر نیست. در عوض یک خط به متن برنامه لیست 2-

1 اضافه شده است. در اولین خط از این برنامه، متنی با عنوان اینکه نام شما چیست؟ بر روی صفحه ظاهر می شود. سپس

در خط بعدی پس از نوشتن کلمه **Hello**، برنامه منتظر دریافت ورودی از کاربر می شود. بدین معنی که این بار تا

زمانیکه کاربر متنی را به عنوان نام خود وارد نکند اجرای برنامه به پیش نخواهد رفت و خط بعدی اجرا نمی شود. این

برنامه روش ایجاد ارتباط از طریق برنامه با کاربر را نمایش می دهد. در این مثال کاربر دقیقاً متوجه می شود که چه زمانی

باید اطلاعات را وارد نماید و این اطلاعات چه باید باشد در حالیکه در مثال قبل چنین نبود. همانگونه که می بینید در این برنامه آرگومان {0} مستقیماً از طریق دستور `Console.ReadLine()` دریافت می شود و بلافاصله در خروجی نمایش داده می شود. دستور `ReadLine()` نیز یکی از متدهای کلاس `Console` است که بوسیله آن رشته ورودی خوانده می شود.

خروجی برنامه فوق به شکل زیر است :

What is your name?:

(سپس برنامه منتظر دریافت متنی از ورودی توسط کاربر می ماند)

(پس از اینکه کاربر رشته ای را وارد کرد و کلید **Enter** را فشار داد، متن زیر نمایش داده می شود.)

Hello, Meysam!

(سپس اجرای برنامه به دستور بعدی منتقل می شود)

Welcome to the C# Persian Tutorial!

خروجی کامل برنامه :

What is your name?:

Hello, Meysam! Welcome to the C# Persian Tutorial!

توجه کنید که `ReadLine()` به عنوان یک متد، مقداری را به سیستم بازمی گرداند. این مقدار در این برنامه به آرگومان {0} برگردانده می شود. این خط از برنامه را می توان طور دیگری هم نوشت :

```
string myName=Console.ReadLine();  
Console.WriteLine("Hello, {0}!",myName);
```

در این حالت ما یک متغیر از نوع رشته با نام `myName` تعریف کرده ایم که مقدار ورودی در آن ذخیره می شود و سپس از این مقدار به عنوان مقداری که {0} می پذیرد استفاده کرده ایم.

در این درس آموختید که ساختار کلی یک برنامه `C#` چگونه است. هر برنامه `C#` از یک کلاس اصلی تشکیل می شود که این کلاس شامل داده ها و متغیرها و متدهایی می باشد. متد آغازین برنامه که برنامه با آن شروع به اجرا می کند، متد `Main()` است. با استفاده از توابع کتابخانه ای می توان به کلاسها و متدهای `C#` دسترسی پیدا کرد. از جمله این توابع `System` بود که یکی از کلاسهای آن `Console` و چند متد این کلاس، متدهای `WriteLine()` و `ReadLine()` بودند.

درس دوم – عبارات، انواع و متغیرها در `C#`

در این درس به معرفی عبارات، انواع و متغیرها در زبان `C#` می پردازیم. هدف از این درس بررسی موارد زیر است :

- آشنایی با متغیرها
- فراگیری انواع (Types) ابتدایی در `C#`

- فراگیری و درک عبارات (Expressions) در C#
- فراگیری نوع رشته‌ای (String) در زبان C#
- فراگیری چگونگی استفاده از آرایه‌ها (Arrays) در زبان C#

متغیرها، به بیان بسیار ساده، مکانهایی جهت ذخیره اطلاعات هستند. شما اطلاعاتی را در یک متغیر قرار می‌دهید و از این اطلاعات بوسیله متغیر در عبارات C# استفاده می‌نمایید. کنترل نوع اطلاعات ذخیره شده در متغیرها بوسیله تعیین کردن نوع برای هر متغیر صورت می‌پذیرد.

C# زبانی بسیار وابسته به انواع است، بطوریکه تمامی عملیاتی که بر روی داده‌ها و متغیرها در این زبان انجام می‌گیرد با دانستن نوع آن متغیر میسر می‌باشد. قوانینی نیز برای تعیین اینکه چه عملیاتی بر روی چه متغیری انجام شود نیز وجود دارد. (بسته به نوع متغیر)

انواع ابتدایی زبان C# شامل: یک نوع منطقی (Boolean) و سه نوع عددی اعداد صحیح (integer)، اعداد اعشاری (Floating points) و اعداد دسیمال (Decimal) می‌باشد. (به انواع Boolean از اینرو منطقی می‌گوییم که تنها دارای دو حالت منطقی صحیح (True) و یا غلط (False) می‌باشند).

مثال 1 – نشان دادن مقادیر منطقی (Boolean)

```
using System;
class Booleans
{
    public static void Main()
    {
        bool content = true;
        bool noContent = false;
        Console.WriteLine("It is {0} that C# Persian provides C# programming
language
        content.", content);
        Console.WriteLine("The statement above is not {0}.", noContent);
    }
}
```

در این مثال، مقادیر منطقی متغیرهای Boolean به عنوان قسمتی از جمله در خروجی نمایش داده می‌شوند. متغیرهای bool تنها می‌توانند یکی از دو مقدار true یا false را داشته باشند، یعنی همانند برخی از زبانهای برنامه‌سازی مشابه، مانند C و یا ++C، مقدار عددی نمی‌پذیرند، زیرا همانگونه که می‌دانید در این دو زبان هر مقدار عددی صحیح مثبت بغیر از صفر به عنوان true و عدد صفر به عنوان false در نظر گرفته می‌شود و در حقیقت نوع bool در این دو زبان نوعی integer می‌باشند. اما در زبان C# انواع bool یکی از دو مقدار true یا false را می‌پذیرند. خروجی برنامه بالا به صورت زیر است:

It is True that C# Persian provides C# programming language content.
The statement above is not False.

جدول زیر تمامی انواع عددی صحیح C#، اندازه آنها و رنج قابل قبول آنها را نشان می دهد.

نوع	اندازه به بیت	رنج قابل قبول
sbyte	8	128- تا 127
byte	8	0 تا 255
short	16	32768- تا 32767
ushort	16	0 تا 65535
int	32	2147483648- تا 2147483647
uint	32	0 تا 4294967295
long	64	9223372036854775808- تا 9223372036854775807
ulong	64	0 تا 18446744073709551615

از این انواع برای محاسبات عددی استفاده می گردد. یک نوع دیگر را نیز می توان در این جدول اضافه نمود و آن نوع **char** است. هر چند شاید از نظر بسیاری از دوستانی که با زبانهای دیگر برنامه سازی کار کرده اند این تقسیم بندی غلط به نظر آید، اما باید گفت که در زبان C# نوع **char** نیز نوع خاصی از انواع عددی است که رنجی بین صفر تا 65535 دارد و اندازه آن نیز 16 بیتی است، اما به جای نمایش دادن مقادیر عددی تنها می تواند بیان کننده یک کاراکتر باشد. در آینده در این مورد بیشتر توضیح خواهم داد.

جدول زیر تمامی انواع عددی اعشاری زبان C# را نمایش می دهد.

نوع	اندازه به بیت	دقت	رنج قابل قبول
float	32	7 رقم	1.5×10^{-45} تا 3.4×10^{38}
double	64	16-15 رقم	5.0×10^{-324} تا 1.7×10^{308}
decimal	128	29-28 رقم دسیمال	1.0×10^{-28} تا 7.9×10^{28}

انواعی از نوع **floating point** هنگامی استفاده می شوند که محاسبات عددی به دقت های اعشاری نیاز داشته باشند. همچنین برای منظوره های تجاری استفاده از نوع **decimal** بهترین گزینه است. این نوع تنها در زبان C# وجود دارد و در زبانهای مشابه به آن نظیر **Java** چنین نوعی در نظر گرفته نشده است.

در یک زبان برنامه‌سازی نتایج بوسیله ایجاد یک سری عبارت تولید می‌گردند. عبارات از ترکیب متغیرها و عملگرها در دستورالعمل‌های یک زبان ایجاد می‌گردند. (توجه نمایید که عبارت معادل **expression** و دستورالعمل معادل **statement** می‌باشد که ایندو با یکدیگر متفاوت می‌باشند.) جدول زیر عملگرهای موجود در زبان **C#**، حق تقدم آنها و شرکت پذیری آنها را نشان می‌دهد.

شرکت پذیری	عملگر(ها)	نوع عمل
از چپ	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	عملیات ابتدایی
از چپ	+ - ! ~ ++x --x (T)x	عملیات یکانی
از چپ	* / %	عملیات ضربی
از چپ	+ -	عملیات جمعی
از چپ	<< >>	عمل شیفت
از چپ	< > <= >= is	عملیات رابطه‌ای
از راست	= = !=	عملیات تساوی
از چپ	&	عمل AND منطقی
از چپ		عمل OR منطقی
از چپ	^	عمل XOR منطقی
از چپ	&&	عمل AND شرطی
از چپ		عمل OR شرطی
از چپ	?:	عمل شرطی
از راست	= *= /= %= += -= <<= >>= &= ^= =	عمل انتساب

شرکت پذیری از چپ بدین معناست که عملیات از چپ به راست محاسبه می‌شوند. شرکت پذیری از راست بدین معناست که تمامی محاسبات از راست به چپ صورت می‌گیرند. به عنوان مثال در یک عمل تساوی، ابتدا عبارات سمت راست تساوی محاسبه شده و سپس نتیجه به متغیر سمت چپ تساوی تخصیص داده می‌شود.

مثال 2- عملگرهای یکانی (Unary)

```
using System;
class Unary
{
    public static void Main()
    {
```

```

int unary = 0;
int preIncrement;
int preDecrement;
int postIncrement;
int postDecrement;
int positive;
int negative;
sbyte bitNot;
bool logNot;
preIncrement = ++unary;
Console.WriteLine("Pre-Increment: {0}", preIncrement);
preDecrement = --unary;
Console.WriteLine("Pre-Decrement: {0}", preDecrement);
postDecrement = unary--;
Console.WriteLine("Post-Decrement: {0}", postDecrement);
postIncrement = unary++;
Console.WriteLine("Post-Increment: {0}", postIncrement);
Console.WriteLine("Final Value of Unary: {0}", unary);
positive = -postIncrement;
Console.WriteLine("Positive: {0}", positive);
negative = +postIncrement;
Console.WriteLine("Negative: {0}", negative);
bitNot = 0;
bitNot = (sbyte)(~bitNot);
Console.WriteLine("Bitwise Not: {0}", bitNot);
logNot = false;
logNot = !logNot;
Console.WriteLine("Logical Not: {0}", logNot);
}
}

```

به هنگام محاسبه عبارات، دو عملگر $x++$ و $x--$ (که در اینجا کاراکتر x بیان کننده آن است که عملگرهای $++$ و $--$ در جلوی عملوند قرار می گیرند **post-increment** و **post-decrement**) ابتدا مقدار فعلی عملوند (**operand**) خود را باز می گرداند و سپس به عملوند خود یک واحد اضافه کرده یا از آن یک واحد می کاهند. عملگر $++$ یک واحد به عملوند خود اضافه می کند و عملگر $--$ یک واحد از عملوند خود می کاهد. بدین ترتیب عبارت $x++$ معادل است با عبارت $x = x + 1$ و یا $x += 1$ اما همانطور که گفته شد باید توجه داشته باشید که این عملگرها ($++$ و $--$) ابتدا مقدار فعلی عملوند خود را برگشت می دهند و سپس عمل خود را روی آنها انجام می دهند. بدین معنی که در عبارت $x = y++$ در صورتیکه در ابتدای اجرای برنامه مقدار $x = 0$ و $y = 1$ باشد، در اولین اجرای برنامه مقدار x برابر با 1 یعنی مقدار y می شود و سپس به متغیر y یک واحد افزوده می شود، در صورتیکه اگر این عبارت را بصورت $x = ++y$ بنویسیم در اولین اجرای برنامه، ابتدا به مقدار متغیر y یک واحد افزوده می شود و سپس این مقدار به متغیر x تخصیص داده می شود که در این حالت مقدار

متغیر x برابر با 2 می‌شود. (در مورد عملگر $-$ نیز چنین است.) پس با این توضیح می‌توان گفت که دو عملگر $++x$ و $-x$ ابتدا به عملوند خود یک واحد اضافه یا یک واحد از آن کم می‌کنند و سپس مقدار آنها را باز می‌گردانند.

در مثال 2، مقدار متغیر **unary** در قسمت اعلان برابر با 0 قرار گرفته است. هنگامیکه از عملگر $++x$ استفاده می‌کنیم، به مقدار متغیر **unary** یک واحد افزوده می‌شود و مقدارش برابر با 1 می‌گردد و سپس این مقدار، یعنی 1، به متغیر **preIncrement** تخصیص داده می‌شود. عملگر $-x$ مقدار متغیر **unary** را به 0 باز می‌گرداند و سپس این مقدار را به متغیر **preDecrement** نسبت می‌دهد.

هنگامیکه از عملگر $x--$ استفاده می‌شود، مقدار متغیر **unary** یا همان مقدار صفر، به متغیر **postDecrement** تخصیص داده می‌شود و سپس از مقدار متغیر **unary** یک واحد کم شده و مقدار این متغیر به 1- تغییر می‌کند. سپس عملگر $x++$ مقدار متغیر **unary**، یعنی همان 1-، را به متغیر **postIncrement** تخصیص می‌دهد و سپس یک واحد به مقدار متغیر **unary** می‌افزاید تا مقدار این متغیر برابر با 0 (صفر) شود.

مقدار متغیر **bitNot** در هنگام اعلان برابر با صفر است. با استفاده از عملگر نقیض بیتی ($\sim$) (یا عملگر مکمل‌گیری) متغیر **bitNot** بعنوان یک بایت در نظر گرفته می‌شود و مقدار آن منفی یا نقیض می‌شود. در عملیات بیتی نقیض بدین معناست که تمامی یک‌ها به صفر و تمامی صفرها به یک تبدیل شوند. در این حالت نمایش باینری عدد صفر یا همان 00000000 به نقیض آن یعنی 11111111 تبدیل می‌گردد.

در این مثال به عبارت $(\sim \text{bitNot})(\text{sbyte})$ توجه نمایید. هر عملی که بر روی انواع **byte**، **unshort**، **short** و **sbyte** انجام شود، مقداری از نوع **int** را باز می‌گرداند. بمنظور اینکه بتوانیم نتیجه دلخواه را به متغیر **bitNot** تخصیص دهیم باید از فرمت **operator (Type)** استفاده نماییم که در آن **Type** نوعی است می‌خواهیم نتیجه ما به آن نوع تبدیل شود و **operator** عملی است که بر روی متغیر صورت می‌پذیرد. به بیان دیگر چون می‌خواهیم مقدار متغیر **bitNot** بصورت بیتی در نظر گرفته شود، پس باید نتیجه عمل ما بصورت بیتی در آن ذخیره شود که استفاده از نوع **sbyte** باعث می‌شود تا نتیجه به فرم بیتی (یا بایتی) در متغیر ما ذخیره شود. باید توجه نمایید که استفاده از فرمت **(Type)** یا در اصطلاح عمل **Casting**، در مواقعی که می‌خواهیم تغییری از یک نوع بزرگتر به نوع کوچکتر ایجاد نماییم، مورد استفاده قرار گیرد، چرا که در این حالات ممکن است با از دست دادن اطلاعات مواجه باشیم. در این مثال چون می‌خواهیم نوع بزرگتر **int** را به (32 بیتی) به نوع کوچکتر **sbyte** (8 بیتی) تبدیل نماییم، بدین منظور باید بطور صریح از عمل **Casting** استفاده نماییم تا اطلاعاتی در این تبدیل از بین نرود. در مورد تبدیل انواع کوچکتر به انواع بزرگتر (مثلاً تبدیل **sbyte** به **int**) نیازی به استفاده از عمل **Casting** نیست چرا که امکان از بین رفتن اطلاعات وجود ندارد. در ضمن باید به یک نکته مهم توجه نمایید و آن تبدیل انواع علامتدار (**Signed**) و بدون علامت (**Unsigned**) به یکدیگر است. در این حالت خطر بسیار مهمی داده‌های شما را تهدید می‌نماید. بحث در مورد مسائل پیچیده‌تر در مورد

تبدیل انواع علامتدار و بدون علامت به یکدیگر در اینجا نمی‌گنجد و سعی می‌کنم تا آنها را در مطالب بعدی و در جای لازم مورد بحث و بررسی قرار دهم. (در صورتیکه برخی از مطالب این قسمتها برای شما به خوبی قابل درک نیست، نگران نباشید چراکه در آینده در مثالهایی که خواهید دید تمامی این مطالب را در عمل نیز حس کرده و با آنها آشنا خواهید شد.)

عملگر بعدی که در این برنامه مورد استفاده قرار گرفته است، عملگر نقیض منطقی یا همان "!" است که امکان تغییر مقدار یک متغیر Boolean را از true به false و بالعکس را فراهم می‌آورد. در مثال بالا (مثال شماره 2) مقدار متغیر logNot پس از استفاده از عملگر "!" از false به true تغییر کرده است. با توجه به توضیحات اخیر خروجی زیر از برنامه مثال 2 مورد انتظار است :

Pre-Increment: 1
Pre-Decrement 0
Post-Decrement: 0
Post-Increment -1
Final Value of Unary: 0
Positive: 1
Negative: -1
Bitwise Not: -1
Logical Not: True

مثال 3 - عملگرهای دوتایی

```
using System;
class Binary
{
    public static void Main()
    {
        int x, y, result;
        float floatResult;
        x = 7;
        y = 5;
        result = x+y;
        Console.WriteLine("x+y: {0}", result);
        result = x-y;
        Console.WriteLine("x-y: {0}", result);
        result = x*y;
        Console.WriteLine("x*y: {0}", result);
        result = x/y;
        Console.WriteLine("x/y: {0}", result);
        floatResult = (float)x/(float)y;
        Console.WriteLine("x/y: {0}", floatResult);
        result = x%y;
        Console.WriteLine("x%y: {0}", result);
        result += x;
        Console.WriteLine("result+=x: {0}", result);
    }
}
```

}

خروجی این برنامه به فرم زیر است :

$x+y$: 12

$x-y$: 2

$x*y$: 35

x/y : 1

x/y : 1.4

$x \% y$: 2

$result += x$: 9

مثال 3 استفاده‌های متفاوتی از عملگرهای دوتایی را نشان می‌دهد. (منظور از عملگر دوتایی، عملگری است که دارای دو عملوند می‌باشد مانند عملگر جمع "+"). بسیاری از عملگرهای مورد استفاده در این مثال عملگرهای ریاضی هستند و نتیجه عمل آنها مشابه عملی است که از آنها در ریاضیات دیده‌اید. از نمونه این عملگرها می‌توان به عملگرهای جمع "+", تفریق "-", ضرب "*", و تقسیم "/" اشاره نمود.

متغیر **floatResult** از نوع اعشاری یا **float** تعریف شده است. در این مثال نیز صریحاً از عمل **Casting** جهت اسفاده از دو متغیر **x** و **y** که از نوع **int** هستند، برای انجام عملی که نتیجه‌اش از نوع **float** است، استفاده کرده‌ایم.

در این مثال از عملگر "%" نیز استفاده کرده‌ایم. این عملگر در عملیات تقسیم کاربرد دارد و باقیمانده تقسیم را برمی‌گرداند. یعنی دو عملوند خود را بر یکدیگر تقسیم می‌کند و باقیمانده این تقسیم را برمی‌گرداند.

در این مثال همچنین فرم جدیدی از عمل انتساب را بصورت **result+=x** مشاهده می‌نمایید. استفاده از عملگرهای انتسابی که خود ترکیبی از دو عملگر هستند، جهت سهولت در امر برنامه‌نویسی مورد استفاده قرار می‌گیرند. عبارت فوق معادل **result = result+x** می‌باشد. یعنی مقدار قبلی متغیر **result** با مقدار متغیر **x** جمع می‌شود و نتیجه در متغیر **result** قرار می‌گیرد.

یکی دیگر از انواعی که تا کنون با آن سر و کار داشته‌ایم نوع رشته‌ای (**string**) است. یک رشته، از قرار گرفتن تعدادی کاراکتر در کنار یکدیگر که داخل یک زوج کوتیشن " " قرار گرفته‌اند، ایجاد می‌گردد. بعنوان مثال **"Hi This is a string type"**. در اعلان متغیرها نیز در صورت تعریف متغیری از نوع رشته‌ای، در صورت نیاز به تخصیص مقدار به آن، حتماً کاراکترهایی که می‌خواهیم بعنوان یک رشته به متغیرمان نسبت دهیم را باید داخل یک زوج کوتیشن " " قرار دهیم. به مثال زیر توجه نمایید.

string Name;

...

Name = "My name is Meysam";

همانطور که در این مثال مشاهده می‌نمایید، متغیری از نوع رشته‌ای تحت نام **Name** تعریف شده است و سپس در جایی از برنامه که نیاز به تخصیص مقدار برای این متغیر وجود دارد، عبارت مورد نظر را داخل دو کوتیشن قرار داده و به متغیر خود تخصیص داده‌ایم. رشته‌ها از پر کاربردترین انواع در زبان‌های برنامه‌سازی جهت ایجاد ارتباط با کاربر و دریافت اطلاعات از کاربر می‌باشند. (همانطور که در درس قبل اول نیز گفته شد، دستور **Console.ReadLine()** یک رشته را

از ورودی دریافت می‌نماید.) در مثالهایی که در طی درسهای این سایت خواهید دید، نمونه‌های بسیاری از کاربرد انواع مختلف و نیز نوع رشته‌ای را خواهید دید.

آرایه‌ها (Arrays)

یکی دیگر از انواع داده‌ای در زبان C#، آرایه‌ها (*Arrays*) می‌باشند. یک آرایه را به عنوان مخزنی برای نگهداری اطلاعات در نظر می‌گیریم که دارای لیستی از محلهایی است که در آنها اطلاعات ذخیره شده است و از طریق این لیست می‌توان به اطلاعات آنها دسترسی پیدا نمود. به هنگام اعلان آرایه‌ها باید نوع، اندازه و تعداد بعد آنها را نیز معین نمود.

مثال 4- آرایه‌ها و عملیات بر روی آنها

```
using System;
class Array
{
    public static void Main()
    {
        int[] myInts = { 5, 10, 15 };
        bool[,] myBools = new bool[2][];
        myBools[0] = new bool[2];
        myBools[1] = new bool[1];
        double[,] myDoubles = new double[2, 2];
        string[] myStrings = new string[3];
        Console.WriteLine("myInts[0]: {0}, myInts[1]: {1}, myInts[2]: {2}",
myInts[0],
        myInts[1], myInts[2]);
        myBools[0][0] = true;
        myBools[0][1] = false;
        myBools[1][0] = true;
        Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]: {1}", myBools[0][0],
myBools[1][0]);
        myDoubles[0, 0] = 3.147;
        myDoubles[0, 1] = 7.157;
        myDoubles[1, 1] = 2.117;
        myDoubles[1, 0] = 56.00138917;
        Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]: {1}",
myDoubles[0, 0], myDoubles[1, 0]);
        myStrings[0] = "Joe";
        myStrings[1] = "Matt";
        myStrings[2] = "Robert";
        Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1}, myStrings[2]: {2}",
myStrings[0], myStrings[1], myStrings[2]);
    }
}
```

}

خروجی مثال 4 بصورت زیر است :

```
myInts[0]: 5, myInts[1]: 10, myInts[2]: 15  
myBools[0][0]: True, myBools[1][0]: True  
myDoubles[0, 0]: 3.147, myDoubles[1, 0]: 56.00138917  
myStrings[0]: Joe, myStrings[1]: Matt, myStrings[2]: Robert
```

در این مثال انواع مختلفی از آرایه‌ها اعلان شده‌اند. در ابتدا یک آرایه تک بعدی، سپس آرایه‌ای دندانه‌دار و در نهایت نیز یک آرایه دو بعدی در این مثال اعلان شده‌اند.

اولین اعلان در این برنامه مربوط به اعلان آرایه تک بعدی **myInts** می‌باشد که از نوع **int** بوده و دارای 3 عضو می‌باشد که تعداد این اعضا با اعلان چند مقدار در داخل { } معین شده است. همانطور که از این اعلان دریافت می‌شود، آرایه تک بعدی بصورت زیر تعریف می‌شود :

```
type[] arrayName;
```

که در آن **type** نوع آرایه و **arrayName** نام آرایه ایست که تعریف می‌نمائیم.

اما در ابتدا گفته شد که به هنگام اعلان آرایه‌ها اندازه آنها نیز باید مشخص شود. برای تعیین اندازه آرایه، یعنی تعداد عناصری که آرایه در خود جای می‌دهد، می‌توان به چند روش عمل نمود. اولین و ساده‌ترین روش که در این مثال نیز آورده شده است، تخصیص مقادیری به آرایه در داخل یک زوج { } است. بسته به نوع آرایه، تعداد عناصری که داخل این زوج { } قرار می‌گیرند، تعداد عناصر آرایه می‌باشند و مقادیر عناصر آرایه نیز همان مقادیری است که داخل { } قرار گرفته است. به عنوان مثال در مثال 4، اولین آرایه ما دارای 3 عنصر است که مقادیر آنها به ترتیب برابر با 5، 10 و 15 می‌باشد.

روش دیگر جهت تعیین اندازه آرایه استفاده از روش تعریف کامل آرایه است که به فرم کلی زیر می‌باشد.

```
type[] arrayName = new type[n];
```

که در این تعریف، استفاده از کلمه کلیدی **new** باعث ایجاد نمونه‌ای جدید از نوع مورد نظر، می‌شود. **n** نیز تعداد عناصر آرایه است که می‌خواهیم آنرا تولید نماییم. در این حالت باید توجه داشت که آرایه‌ای تهی را تولید نموده‌ایم و هیچ عنصری را در آرایه جای نداده‌ایم و در برنامه باید آرایه را مقدار دهی نماییم. به مثال زیر توجه کنید.

```
int[] myArray = new int[15];
```

این مثال آرایه‌ای تک بعدی از نوع **int** را با اندازه 15 عنصر تولید می‌نماید. یعنی این آرایه قادر است تا 15 عنصر از نوع **int** را در خود ذخیره نماید.

گونه دیگری از آرایه‌ها، آرایه‌های چند بعدی (**Multi Dimensional Arrays**) هستند که برای نگهداری اطلاعات از چندین بعد استفاده می‌کنند و بیشتر برای نگه‌داری جداول و ماتریسها مورد استفاده قرار می‌گیرند. فرم کلی اعلان این آرایه‌ها بصورت زیر است :

```
type[ , , ... , ] arrayName = new type[n1, n2, .... , nm];
```


که در آن تعداد ابعاد آرایه با ویرگول مشخص شده و **n1** تا **nm** نیز تعداد عناصر هر یک از ابعاد است. بعنوان مثال تعریف یک آرایه سه بعدی به فرم زیر است :

```
char[ , , ] charArray = new char[3,5,7];
```

در این مثال یک آرایه سه بعدی از نوع **char** تولید کرده‌ایم که ابعاد آن به ترتیب دارای 3، 5 و 7 عنصر می‌باشند. نوع دیگری از آرایه‌ها، آرایه‌های دندانه‌دار (**Jagged Arrays**) هستند. این نوع آرایه‌ها تنها در زبان **C#** وجود دارند و در صرفه‌جویی حافظه بسیار موثر می‌باشند. یک آرایه دندانه‌دار، در حقیقت یک آرایه تک بعدی است که هر یک از اعضای آن خود یک آرایه تک بعدی می‌باشند. اندازه این عناصر می‌تواند متفاوت باشد و تفاوت این آرایه‌ها با آرایه‌های چند بعدی در همین جا نمایان می‌شود. استفاده از این آرایه‌ها در مواردی کاربرد دارد که نیازی نیست تا تمامی ابعاد آرایه دارای تعداد عناصر مساوی باشند. بعنوان مثال فرض کنید می‌خواهید آرایه‌ای جهت نگهداری تعداد روزهای ماههای مختلف سال تهیه کنید. در صورتیکه بخواهید از آرایه چند بعدی استفاده نمایید، چون تعداد روزهای تمامی ماههای سال یکسان نیست، مجبورید تا تعداد عناصر تمام بعدهای آرایه را برابر با بزرگترین تعداد روز ماهها، یعنی 31، تعریف نمایید. ولی چون تنها 6 ماه دارای 31 روز می‌باشند، برای 6 ماه دیگر تعدادی از عناصر آرایه هیچگاه مورد استفاده قرار نمی‌گیرند و حافظه را به هدر داده‌ایم. اما در صورتیکه برای این مثال از آرایه‌های دندانه‌دار استفاده نماییم، می‌توانیم یک آرایه دندانه‌دار 12 عنصری تعریف نماییم و سپس تعداد عناصر هر یک از اعضای آنرا برابر با تعداد روزهای ماه مورد نظر تعریف کنیم :

با استفاده از آرایه چند بعدی :

```
int[ , ] monthArray = new int[12,31];
```

با استفاده از آرایه دندانه‌دار :

```
int[][] monthArray = new int[12][];
```

در تعریف اول که در آن از آرایه چند بعدی استفاده کردیم، مشاهده می‌کنید که آرایه‌ای دو بعدی تعریف کرده‌ایم که بعد اول آن 12 عضو و بعد دوم آن 31 عضو دارد. این عمل دقیقاً همانند ایجاد یک جدول برای نگهداری روزهای ماههای سال است.

اما در حالت دوم که در آن از آرایه دندانه‌دار بهره برده‌ایم، یک آرایه تعریف نموده‌ایم که بعد اول آن 12 عضو دارد ولی بعد دوم آنرا را تعریف نکرده‌ایم که دارای چند عضو است و هر یک از عناصر بعد اول آرایه می‌تواند دارای تعداد اعضای متفاوتی باشد که با استفاده از این روش می‌توان به هر یک از ماههای سال تعداد روزهای مورد نظر آن ماه را تخصیص داد و فضایی بلا استفاده ایجاد نخواهیم کرد. توجه نمایید که چون تعداد عناصر بعد دیگر این آرایه معین نشده است در برنامه باید این تعداد عنصر را مشخص نماییم :

```
monthArray[1] = new int[31];
```

```
monthArray[10] = new int [30];
```

مشاهده می‌کنید که به هر ماه، تعداد عنصر مورد نیاز خود را تخصیص داده‌ایم. تنها باید به تفاوت اعلان آرایه‌های دندانه‌دار با آرایه‌های چند بعدی توجه نمایید.

دسترسی به عناصر آرایه از طریق اندیس امکان پذیر است. اندیس شماره محل ذخیره سازی داده های ما می باشد که با دادن این شماره می توانیم به داده مورد نظر دسترسی پیدا کنیم. در **C#** همانند **C** و **C++** اندیس خانه های آرایه از صفر آغاز می گردد یعنی اولین خانه آرایه دارای شماره صفر است و عناصر بعدی به ترتیب یک واحد به اندیسشان اضافه می گردد. پس شماره اندیس آرایه همیشه یک واحد کمتر از تعداد عناصر آن است، یعنی آرایه ای که 10 عضو دارد بزرگترین اندیس خانه هایش 9 می باشد. دسترسی به عناصر هر یک از ابعاد آرایه با اندیس امکان پذیر است. معمولاً به بعد اول آرایه سطر و به بعد دوم آن ستون می گوید. مثلاً **monthArray[3,7]** عنصر واقع در سطر 3 و ستون 7 آرایه را مشخص می نماید. (توجه داشته باشید که اندیس دهی آرایه از صفر آغاز می شود و بعنوان مثال **intArray[12]** به خانه شماره 12 آرایه اشاره می کند اما فراموش نکنید چون اندیس آرایه از صفر آغاز می شود خانه شماره 12 آرایه، سیزدهمین داده شما را در خود جای می دهد).

0	1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	---	----	----	----

اگر شکل فوق را آرایه ای تک بعدی تصور نمایید، مشاهده می نمایید که خانه شماره 5 آرایه حاوی اطلاعات مربوط به ششمین داده ورودی شما می باشد.

نکته دیگری که باید در مورد تعریف آرایه های این مثال متذکر شوم در مورد آریه هایست که از نوع **string** تعریف می شوند. دوستانی که با زبان **C** کار کرده اند حتماً می دانند که آرایه ای از نوع رشته ای در **C** وجود ندارد و برای نگهداری چندین رشته در یک آرایه باید از آرایه دو بعدی استفاده کرد. در **C#** این قابلیت فراهم شده تا با استفاده از یک آرایه تک بعدی بتوان چندین رشته را ذخیره نمود بدین صورت که هر یک از عناصر آرایه یک بعدی محلی برای ذخیره سازی یک رشته است و همانند زبان **C** نیاز به پردازش های گاه پیچیده بر روی آرایه های چند بعدی بمنظور کار با رشته ها، وجود ندارد. بعنوان یک توضیح کمی اختصاصی عرض می کنم که در زبان هایی مانند **C**، در صورتیکه می خواستید چندین رشته را در آرایه ای ذخیره کنید تا بتوانید با اندیس به آنها دسترسی داشته باشید، مجبور به تعریف یک آرایه دو بعدی بودید که با استفاده از تنها اندیس اول آرایه می توانستید به عناصر رشته ای آرایه دسترسی پیدا کنید، اما در **C#** تنها با استفاده از یک آرایه تک بعدی می توان همان کار را انجام داد.

```
string[] stringArray = {"My name is Meysam", "This is C# Persian Blog"}
```

```
.....
```

```
Console.WriteLine("{0}",stringArray[0]);
```

```
.....
```

همانطور که در این مثال ملاحظه می کنید، آرایه ای از نوع رشته تعریف شده و دو عنصر به آن تخصیص داده شده است و در جایی در متن برنامه با استفاده از اندیس از اولین عنصر این آرایه برای نمایش در خروجی استفاده گردیده است. خروجی این برنامه به شکل زیر است :

```
My name is Meysam
```

مطلب این درس در اینجا به پایان می‌رسد. در صورتیکه نیاز دارید تا در مورد عملگرهای زبان C# بیشتر توضیح دهم حتماً ذکر کنید تا در مطلب بعدی توضیح کاملتری در مورد آنها برای شما تهیه کنم. خیلی دوست داشتم که در مورد تمام عملگرهای زبان C# در همین درس توضیح بدهم اما هم فرصت اندک است و هم حجم مطلب این قسمت زیاد می‌شد و هم اینکه فکر کردم احتمالاً دوستان با این عملگرها آشنایی دارند. نکته دیگری که باید به آن اشاره کنم و اینست که در این سایت سعی شده است تا زبان برنامه‌نویسی C# به سادگی و به سرعت آموزش داده شود و علت اینکه به برخی از جزئیات تخصصی پرداخته نمی‌شود نیز همین مطلب می‌باشد. در آینده در مورد آرایه‌ها بیشتر صحبت می‌کنم چون عناصر مفید و سودمندی هستند. امید است پس از کامل کردن مطالب مقدماتی در این سایت و با همکاری شما عزیزان بتوانم به مطالب پیشرفته‌تری از زبان C# بپردازم. بیان نظرات و پیشنهادات شما چه در زمینه مطالب ارائه شده و چه در زمینه متن ارائه شده به شما از لحاظ سادگی و روانی در درک، مرا در امر بهبود مطالب یاری می‌نماید.

درس سوم - دستورالعمل‌های کنترلی و شرطی

در این درس با دستورالعمل‌های کنترل و انتخاب در C# آشنا می‌شوید. هدف این درس عبارتست از :

- یادگیری دستور `if`
- یادگیری دستور `switch`
- نحوه بکارگیری دستور `break` در دستور `switch`
- درک صحیح از نحوه بکارگیری دستور `goto`

بررسی دستور `if` و انواع مختلف آن

در درسهای گذشته، برنامه‌هایی که مشاهده می‌کردید از چندین خط دستور تشکیل شده بودند که یکی پس از دیگری اجرا می‌شدند و سپس برنامه خاتمه می‌یافت. در این برنامه‌ها هیچ عمل تصمیم‌گیری صورت نمی‌گرفت و تنها دستورات برنامه به ترتیب اجرا می‌شدند. مطالب این درس نحوه تصمیم‌گیری در یک برنامه را به شما نشان می‌دهد.

اولین دستور تصمیم‌گیری که ما آنرا بررسی می‌نماییم، دستورالعمل `if` است. این دستور دارای سه فرم کلی : تصمیم‌گیری ساده، تصمیم‌گیری دوگانه، تصمیم‌گیری چندگانه می‌باشد.

مثال 3-1 - فرم‌های دستورالعمل `if`

using System;

```
class IfSelect
{
    public static void Main()
    {
```

```

string myInput;
int myInt;
Console.WriteLine("Please enter a number: ");
myInput = Console.ReadLine();
myInt = Int32.Parse(myInput);
// تصمیم‌گیری ساده و اجرای عمل داخل دو گروهه
if (myInt > 0)
{
    Console.WriteLine("Your number {0} is greater than zero.", myInt);
}
// تصمیم‌گیری ساده و اجرای عمل بدون استفاده از دو گروهه
if (myInt < 0)
    Console.WriteLine("Your number {0} is less than zero.", myInt);
// تصمیم‌گیری دوگانه
if (myInt != 0)
{
    Console.WriteLine("Your number {0} is not equal to zero.", myInt);
}
else
{
    Console.WriteLine("Your number {0} is equal to zero.", myInt);
}
// تصمیم‌گیری چندگانه
if (myInt < 0 || myInt == 0)
{
    Console.WriteLine("Your number {0} is less than or equal to zero.", myInt);
}
else if (myInt > 0 && myInt <= 10)
{
    Console.WriteLine("Your number {0} is between 1 and 10.", myInt);
}
else if (myInt > 10 && myInt <= 20)
{
    Console.WriteLine("Your number {0} is between 11 and 20.", myInt);
}
else if (myInt > 20 && myInt <= 30)
{
    Console.WriteLine("Your number {0} is between 21 and 30.", myInt);
}
else
{
    Console.WriteLine("Your number {0} is greater than 30.", myInt);
}
} //Main() پایان متد

```

پایان کلاس //IfSelect }

برنامه 1-3 از یک متغیر myInt برای دریافت ورودی از کاربر استفاده می‌نماید، سپس با استفاده از یک سری دستورات کنترلی، که همان دستور if در اینجا است، عملیات خاصی را بسته به نوع ورودی انجام می‌دهد. در ابتدای این برنامه عبارت Please enter a number: در خروجی چاپ می‌شود. دستور Console.ReadLine() منتظر می‌ماند تا کاربر ورودی وارد کرده و سپس کلید Enter را فشار دهد. همانطور که در قبل نیز اشاره کرده‌ایم، دستور Console.ReadLine() عبارت ورودی را به فرم رشته دریافت می‌نماید پس مقدار ورودی کاربر در اینجا که یک عدد است به فرم رشته‌ای در متغیر myInput که از نوع رشته‌ای تعریف شده است قرار می‌گیرد. اما میدانیم که برای اجرای محاسبات و یا تصمیم‌گیری بر روی اعداد نمی‌توان از آنها در فرم رشته‌ای استفاده کرد و باید آنها را بصورت عددی مورد استفاده قرار داد. به همین منظور باید متغیر myInput را به نحوی به مقدار عددی تبدیل نماییم. برای این منظور از عبارت Int32.Parse() استفاده می‌نماییم. این دستور مقدار رشته‌ای متغیر داخل پرانتز را به مقدار عددی تبدیل کرده و آنرا به متغیر دیگری از نوع عددی تخصیص می‌دهد. در این مثال نیز همانطور که دیده می‌شود، myInput که از نوع رشته‌ای است در داخل پرانتز قرار گرفته و این مقدار برابر با myInt که از نوع int است قرار گرفته است. با این کار مقدار عددی رشته ورودی کاربر به متغیر myInt تخصیص داده می‌شود. (توضیح کامل‌تری در مورد Int32 و سایر تبدیلات مشابه به آن در درس‌های آینده و در قسمت نوع‌های پیشرفته مورد بررسی قرار می‌گیرند.) حال ما متغیری از نوع مورد نظر در دست داریم و می‌توانیم با استفاده از دستور if بر روی آن پردازش انجام داده و تصمیم‌گیری نماییم.

دستور if

اولین دستور بصورت *if (boolean expression) {statements}* آورده شده است. دستور if با استفاده از کلمه کلیدی if آغاز می‌شود. سپس یک عبارت منطقی درون یک زوج پرانتز قرار می‌گیرد. پس از بررسی این عبارات منطقی دستورالعمل/دستورالعمل‌های داخل کروشه اجرا می‌شوند. همانطور که مشاهده می‌نمایید، دستور if یک عبارت منطقی را بررسی می‌کند. در صورتیکه مقدار این عبارات true باشد دستورالعمل‌های داخل بلوک خود را اجرا می‌نماید (قبلاً توضیح داده شد که دستورالعمل‌هایی که داخل یک زوج کروشه {} قرار می‌گیرند در اصطلاح یک بلوک نامیده می‌شوند.) و در صورتیکه مقدار آن برابر با false باشد اجرای برنامه به بعد از بلوک if منتقل می‌شود. در این مثال همانطور که ملاحظه می‌نمایید، عبارت منطقی دستور if به شکل `if(myInt > 0)` است. در صورتیکه مقدار myInt بزرگتر از عدد صفر باشد، دستور داخل بلوک if اجرا می‌شود و در غیر اینصورت اجرای برنامه به بعد از بلوک if منتقل می‌گردد.

دومین دستور if در این برنامه بسیار شبیه به دستور اول است، با این تفاوت که در این دستور، دستور اجرایی if درون یک بلوک قرار نگرفته است. در صورتیکه بخواهیم با استفاده از دستور if تنها یک دستورالعمل اجرا شود، نیازی به

استفاده از بلوک برای آن دستورالعمل نمی‌باشد. استفاده از بلوک تنها زمانی ضروری است که بخواهیم از چندین دستور استفاده نماییم.

دستور *if-else*

در بیشتر موارد از تصمیم‌گیری‌های دوگانه یا چندگانه استفاده می‌شود. در این نوع تصمیم‌گیری‌ها، دو یا چند شرط مختلف بررسی می‌شوند و در صورت *true* بودن یکی از آنها عمل مربوط به آن اجرا می‌گردد. سومین دستور *if* در این برنامه نشان‌دهنده یک تصمیم‌گیری دوگانه است. در این حالت در صورتیکه عبارت منطقی دستور *if* برابر با *true* باشد دستور بعد از *if* اجرا می‌شود و در غیر اینصورت دستور بعد از *else* به اجرا در می‌آید. در حقیقت در این حالت می‌گوئیم "اگر شرط *if* صحیح است دستورات مربوط به *if* را انجام بده و در غیر اینصورت دستورات *else* را اجرا کن".

فرم کلی دستور *if-else* بصورت زیر است :

```
if (boolean expression)  
    {statements}
```

```
else  
    {statements}
```

که در آن *boolean expression* عبارت منطقی است که صحت آن مورد بررسی قرار می‌گیرد و *statements* دستور یا دستوراتی است که اجرا می‌گردند.

دستور *if-else if ... else* یا *if* تودرتو

در صورتیکه نیاز باشد تا چندین حالت منطقی مورد بررسی قرار گیرد و دستورات مربوط به یکی از آنها اجرا شود، از فرم تصمیم‌گیری چندگانه استفاده می‌نماییم. این نوع استفاده از دستور *if* در اصطلاح به *if* تودرتو (*Nested If*) معروف است چراکه در آن از چندین دستور *if* مرتبط به یکدیگر استفاده شده است. چهارمین دستور *if* در مثال 1-3 استفاده از *if* تودرتو را نشان می‌دهد. در این حالت نیز دستور با کلمه کلیدی *if* آغاز می‌گردد. شرطی بررسی شده و در صورت *true* بودن دستورات مربوط به آن اجرا می‌گردد. اما اگر مقدار این عبارت منطقی *false* بود آنگاه شرطهای فرعی دیگری بررسی می‌شوند. این شرطهای فرعی با استفاده از *else if* مورد بررسی قرار می‌گیرند. هر یک از این شرطها دارای عبارات منطقی مربوط به خود هستند که در صورت *true* بودن عبارت منطقی دستورات مربوط به آنها اجرا می‌گردد و در غیر اینصورت شرط بعدی مورد بررسی قرار می‌گیرد. باید توجه کنید که در ساختار *if* تودرتو تنها یکی از حالتها اتفاق می‌افتد و تنها یکی از شرطها مقدار *true* را باز می‌گرداند.

فرم کلی *if* تودرتو بشکل زیر است :

```
if (boolean expression)  
    {statements}
```

```
else if (boolean expression)  
    {statements}
```

```
...
```

```
else  
    {statements}
```

عملگرهای OR و AND (|| و &&)

نکته دیگری که باید در اینجا بدان اشاره کرد، نوع شرطی است که در عبارت منطقی دستور if آخر مورد استفاده قرار گرفته است. در این عبارت منطقی از عملگر || استفاده شده است که بیانگر OR منطقی است. عملگر OR زمانی مقدار true بازمی‌گرداند که حداقل یکی از عملوندهای آن دارای مقدار true باشد. بعنوان مثال در عبارت (myInt < 0 || myInt == 0)، در صورتیکه مقدار متغیر myInt کوچکتر یا مساوی با صفر باشد، مقدار عبارت برابر با true است. نکته قابل توجه آنست که در زبان C#، همانطور که در [درس دوم](#) به آن اشاره شد، دو نوع عملگر OR وجود دارد. یکی OR منطقی که با || نمایش داده می‌شود و دیگری OR معمولی که با && نشان داده می‌شود. تفاوت بین این دو نوع OR در آنست که OR معمولی هر دو عملگر خود را بررسی می‌نماید اما OR منطقی تنها در صورتیکه عملگر اول آن مقدار false داشته باشد به بررسی عملگر دوم خود می‌پردازد.

عبارت منطقی (myInt > 0 && myInt <= 10) حاوی عملگر AND شرطی (&&) می‌باشد. این عبارت در صورتی مقدار true بازمی‌گرداند که هر دو عملوند AND دارای مقدار true باشند. یعنی در صورتیکه myInt هم بزرگتر از صفر باشد و هم کوچکتر از 10، مقدار عبارت برابر با true می‌گردد. در مورد AND نیز همانند OR دو نوع عملگر وجود دارد. یکی AND معمولی (&) و دیگری AND شرطی (&&). تفاوت این دو نیز در آنست که AND معمولی (&) همیشه هر دو عملوند خود را بررسی می‌نماید ولی AND شرطی (&&) تنها هنگامی به بررسی عملوند دوم خود می‌پردازد که مقدار اولین عملوندش برابر با true باشد. عملگرهای منطقی || و && را در اصطلاح عملگرهای میانبر (short-circuit) می‌نامند چراکه تنها در صورت لزوم عملوند دوم خود را بررسی می‌نمایند و از اینرو سریعتر اجرا می‌شوند.

بررسی دستور switch

همانند دستور if، دستور switch نیز امکان تصمیم‌گیری را در یک برنامه فراهم می‌نماید.

مثال 2-3 - دستورالعمل switch

```
using System;
```

```
class SwitchSelect
```

```
{  
    public static void Main()  
    {  
        string myInput;  
        int myInt;  
        begin:  
        Console.WriteLine("Please enter a number between 1 and 3: ");  
        myInput = Console.ReadLine();  
        myInt = Int32.Parse(myInput);  
        // دستور switch به همراه متغیری از نوع صحیح
```

```

switch (myInt)
{
    case 1:
        Console.WriteLine("Your number is {0}.", myInt);
        break;
    case 2:
        Console.WriteLine("Your number is {0}.", myInt);
        break;
    case 3:
        Console.WriteLine("Your number is {0}.", myInt);
        break;
    default:
        Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);
        break;
} //پایان بلوک switch

decide:
Console.Write("Type \"continue\" to go on or \"quit\" to stop: ");
myInput = Console.ReadLine();
//دستور switch به همراه متغیری از نوع رشته ای
switch (myInput)
{
    case "continue":
        goto begin;
    case "quit":
        Console.WriteLine("Bye.");
        break;
    default:
        Console.WriteLine("Your input {0} is incorrect.", myInput);
        goto decide;
} //پایان بلوک switch

} //پایان متد Main()

} //پایان کلاس SwitchSelect

```

مثال 2-3 دو مورد استفاده از دستور switch را نشان می‌دهد. دستور switch بوسیله کلمه کلیدی switch آغاز شده و به دنبال آن عبارت دستور switch قرار می‌گیرد. عبارت دستور switch می‌تواند یکی از انواع زیر باشد: short, ushort, int, uint, long, ulong, char, string, enum, sbyte, byte. (نوع enum در مبحث جداگانه‌ای مورد بررسی قرار خواهد گرفت). در اولین دستور switch در مثال 2-3، عبارت دستور switch از نوع عددی صحیح (int) می‌باشد.

به دنبال دستور و عبارت switch، بلوک switch قرار می‌گیرد که در آن گزینه‌هایی قرار دارند که جهت منطبق بودن با مقدار عبارت switch مورد بررسی قرار می‌گیرند. هر یک از این گزینه‌ها با استفاده از کلمه کلیدی case

مشخص می‌شوند. پس از کلمه کلیدی `case` خود گزینه قرار می‌گیرد و به دنبال آن ":" و سپس دستوری که باید اجرا شود. بعنوان مثال به اولین دستور `switch` در این برنامه توجه نمایید. در اینجا عبارت دستور `switch` از نوع `int` است. هدف از استفاده از دستور `switch` آنست که از بین گزینه‌های موجود در بلوک `switch`، گزینه‌ای را که مقدارش با مقدار عبارت `switch` برابر است پیدا شده و عمل مرتبط با آن گزینه اجرا شود. در این مثال مقدار متغیر `myInt` بررسی می‌شود. سپس اگر این مقدار با یکی از مقادیر گزینه‌های داخل بلوک `switch` برابر بود، دستور یا عمل مربوط به آن گزینه اجرا می‌گردد. توجه نمایید که در این مثال منظور ما از گزینه همان عدد پس از `case` است و منظور از دستور عبارتی است که پس از ":" قرار گرفته است. بعنوان مثال، در دستور زیر :

case 1:

```
Console.WriteLine("Your number is {0}.", myInt);
```

عدد 1، گزینه مورد نظر ما و دستور `Console.WriteLine(...)`، عمل مورد نظر است. در صورتیکه مقدار `myInt` برابر با عدد 1 باشد آنگاه دستور مربوط به `case 1` اجرا می‌شود که همان `Console.WriteLine("Your number is {0}.", myInt);` است. پس از منطبق شدن مقدار عبارت `switch` با یکی از `case` ها، بلوک `switch` باید خاتمه یابد که این عمل بوسیله استفاده از کلمه کلیدی `break`، اجرای برنامه را به اولین خط بعد از بلوک `switch` منتقل می‌نماید.

همانطور که در ساختار دستور `switch` مشاهده می‌نمایید، علاوه بر `case` و `break`، دستور دیگری نیز در داخل بلوک وجود دارد. این دستور یعنی `default`، برای زمانی مورد استفاده قرار می‌گیرد که هیچ یک از گزینه‌های بلوک `switch` با عبارت دستور `switch` منطبق نباشند. به عبارت دیگر در صورتیکه مقدار عبارت `switch` با هیچ یک از گزینه‌های `case` برابر نباشد، دستور مربوط به `default` اجرا می‌گردد. استفاده از این دستور در ساختار بلوک `switch` اختیاری است. همچنین قرار دادن دستور `break` پس از دستور `default` نیز اختیاری می‌باشد.

همانطور که قبلاً نیز گفته شد پس از هر دستور `case`، به منظور خاتمه دادن اجرای بلوک `switch` باید از یک `break` استفاده نمود. دو استثنا برای این موضوع وجود دارد. اول اینکه دو دستور `case` بدون وجود کد و دستورات عملی در بین آنها، پشت سر هم قرار گیرند و دیگری در زمانیکه از دستور `goto` استفاده شده باشد. در صورتیکه دو دستور `case` بدون وجود کدی در بین آنها، پشت سر یکدیگر قرار گیرند، بدین معناست که برای هر دو `case` مورد نظر یک عمل خاص در نظر گرفته شده است. به مثال زیر توجه نمایید.

switch (myInt)

```
{
    case 1:
    case 2:
    case 3:
        Console.WriteLine("Your number is {0}.", myInt);
        break;
    default:
        Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);
        break;
```

}
در این مثال، همانطور که مشاهده می‌کنید، سه دستور case بدون وجود کدی در بین آنها پشت سر یکدیگر قرار گرفته‌اند. این عمل بدین معناست که برای تمامی گزینه‌های 1، 2 و 3 دستور Console.WriteLine("Your number is {0}.", myInt); اجرا خواهد شد. یعنی اگر مقدار myInt برابر با هر یک از مقادیر 1، 2 و 3 باشد، یک دستور برای آن اجرا می‌شود.

نکته قابل توجه دیگر در مورد بلوک switch آنست که، دستورات case حتماً نباید یک دستور باشد بلکه می‌توان از یک بلوک دستور برای case استفاده نمود.

دومین استفاده از دستور switch در مثال 2-3، دارای عبارتی از نوع رشته‌ایست. در این بلوک switch چگونگی استفاده از دستور goto نیز نشان داده شده است. دستور goto اجرای برنامه را به برچسبی (label) که معین شده هدایت می‌نماید. در حین اجرای این برنامه، اگر کاربر رشته continue وارد نماید، این رشته با یکی از گزینه‌های دومین switch منطبق می‌شود. چون دستور case مربوط به این گزینه دارای دستور goto است، اجرای برنامه به برچسبی که این دستور مشخص کرده فرستاده می‌شود، بدین معنی که اجرای برنامه به ابتدای جایی می‌رود که عبارت begin: در آنجا قرار دارد (در اوایل متد Main()). بدین صورت اجرای برنامه از بلوک switch خارج شده و به ابتدای برنامه و در جاییکه برچسب begin: قرار گرفته ارسال می‌شود. در این برنامه، استفاده از چنین حالتی استفاده از goto باعث ایجاد یک حلقه شده است که با وارد کردن عبارت quit اجرای آن به پایان می‌رسد.

در صورتیکه هیچ یک از عبارات continue و یا quit وارد نشوند، اجرای switch به گزینه default می‌رود و در این گزینه ابتدا پیغام خطایی بر کنسول چاپ شده و سپس با استفاده از دستور goto پرشی به برچسب decide صورت می‌گیرد. پس از پرش به برچسب decide، از کاربر پرسیده می‌شود که آیا می‌خواهد اجرای برنامه را ادامه دهد یا خیر. (با وارد کردن گزینه‌های continue یا quit) همانطور که می‌بینید در اینجا نیز حلقه‌ای تولید شده است.

استفاده از دستور goto در بلوک switch می‌تواند موثر باشد اما باید توجه نمایید که استفاده‌های بی مورد از دستور goto باعث ناخوانا شدن برنامه شده و عیب‌یابی (Debug) برنامه را بسیار دشوار می‌نماید. در برنامه‌نویسی‌های امروزی استفاده از دستور goto بغیر از موارد بسیار لازم و ضروری منسوخ شده و به هیچ عنوان توصیه نمی‌شود. برای تولید و ساخت حلقه نیز دستورات مفید و سودمندی در زبان تعبیه شده‌اند که استفاده از goto را به حداقل می‌رسانند. دستورات حلقه در مبحث آینده مورد بررسی قرار خواهند گرفت.

نکته پایانی این مبحث آنست که توجه نمایید که به جای استفاده از دستور switch می‌توانید از چندین دستور if-else استفاده نمایید. دو قطعه برنامه زیر معادل یکدیگر می‌باشند.

```
switch(myChar)
{
    case 'A' :
        Console.WriteLine("Add operation\n");
        break;
```



```

case 'M' :
    Console.WriteLine("Multiple operation\n");
    break;
case 'S' :
    Console.WriteLine("Subtraction operation\n");
    break;
default :
    Console.WriteLine("Error, Unknown operation\n");
    break;
}

```

معادل بلوک switch با استفاده از if-else

```

if (myChar == 'A')
    Console.WriteLine("Add operation\n");
else if (myChar == 'M')
    Console.WriteLine("Multiple operation\n");
else if (myChar == 'S')
    Console.WriteLine("Subtraction operation\n");
else
    Console.WriteLine("Error, Unknown operation\n");

```

همانطور که ملاحظه می‌کنید استفاده از بلوک دستور switch بسیار ساده‌تر از استفاده از if-else های تودرتو است.

در این درس با نحوه تصمیم‌گیری در برنامه بوسیله دستور if و switch آشنا شدید. با نحوه عملکرد و استفاده دستور goto نیز آشنایی پیدا کردید. در پایان مجدداً یادآوری می‌کنم که در استفاده از دستور goto با احتیاط عمل نمایید و به جز در موارد ضروری از آن استفاده نکنید.

درس چهارم – دستورالعمل‌های کنترلی، حلقه‌ها

در این درس نحوه استفاده از دستورالعمل‌های کنترل حلقه در زبان C# را فرا خواهید گرفت. هدف این درس فهم و درک موارد زیر می‌باشد :

ü حلقه while

ü حلقه do -while

ü حلقه for

ü حلقه foreach

ü مطالب تکمیلی درباره دستورالعمل break

ü فراگیری نحوه بکارگیری دستورالعمل continue

در درس قبل، نحوه ایجاد یک حلقه بسیار ساده را با استفاده از دستور **goto** را فرا گرفتید. در همان مطلب نیز اشاره کردیم که این روش، روش مناسبی جهت ایجاد حلقه نیست. در این درس با نحوه صحیح ایجاد حلقه‌ها در زبان **C#** آشنا خواهید شد. اولین دستوری که با آن آشنا می‌شوید نیز دستور **while** است.

حلقه **while**

ابتدا به مثال زیر توجه نمایید.

```
using System;
```

```
class WhileLoop
```

```
{  
    public static void Main()  
    {  
        int myInt = 0;  
        while (myInt < 10)  
        {  
            Console.Write("{0} ", myInt);  
            myInt++;  
        }  
        Console.WriteLine();  
    }  
}
```

مثال **4-1** که در بالا ملاحظه می‌کنید، یک حلقه **while** ساده را نشان می‌دهد. این حلقه با کلمه کلیدی **while** آغاز شده و سپس به دنبال آن یک عبارت منطقی قرار می‌گیرد و مورد بررسی قرار می‌گیرد. تمامی دستورالعمل‌های کنترلی از یک عبارت منطقی بهره می‌گیرند و این بدین معناست که ابتدا این عبارت باید بررسی شود تا مشخص شود مقدار این عبارت **true** است یا **false**. در این مثال مقدار متغیر **myInt** مورد بررسی قرار می‌گیرد تا چک شود آیا مقدارش از 10 کوچکتر هست یا خیر. چون در ابتدای برنامه به این متغیر مقدار صفر تخصیص داده شده است، عبارت منطقی مقدار **true** را باز می‌گرداند و سپس بلوک قرار گرفته بعد از عبارت منطقی مورد اجرا قرار می‌گیرد.

درون بلوک **while** ابتدا مقدار متغیر **myInt** در کنسول نمایش داده می‌شود و سپس یک واحد به مقدار این متغیر افزوده می‌گردد. پس از اتمام بلوک **while** عبارت منطقی مجدداً کنترل می‌شود و در صورتیکه این عبارت مقدار **true** بازگرداند، حلقه **while** مجدداً اجرا می‌شود. زمانیکه عبارت منطقی مقدار **false** برگرداند، اجرا برنامه به اولین دستور بعد از بلوک **while** منتقل می‌شود. در این مثال اعداد صفر تا 9 بر روی صفحه نمایش داده می‌شوند و سپس یک خط خالی چاپ شده و اجرای برنامه خاتمه می‌یابد.

حلقه بعدی که بسیار شبیه به حلقه **while** می‌باشد، حلقه **do-while** است.

حلقه **do-while**

```
using System;
```

```
class DoLoop
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        string myChoice;
```

```
        do
```

```
        {
```

```
            // منویی نمایش داده می شود
```

```
            Console.WriteLine("My Address Book\n");
```

```
            Console.WriteLine("A - Add New Address");
```

```
            Console.WriteLine("D - Delete Address");
```

```
            Console.WriteLine("M - Modify Address");
```

```
            Console.WriteLine("V - View Addresses");
```

```
            Console.WriteLine("Q - Quit\n");
```

```
            Console.WriteLine("Choice (A,D,M,V,or Q): ");
```

```
            // ورودی کاربر بررسی می شود
```

```
            myChoice = Console.ReadLine();
```

```
            // تصمیمی بر اساس ورودی کاربر گرفته می شود
```

```
            switch(myChoice)
```

```
            {
```

```
                case "A":
```

```
                case "a":
```

```
                    Console.WriteLine("You wish to add an address.");
```

```
                    break;
```

```
                case "D":
```

```
                case "d":
```

```
                    Console.WriteLine("You wish to delete an address.");
```

```
                    break;
```

```
                case "M":
```

```
                case "m":
```

```
                    Console.WriteLine("You wish to modify an address.");
```

```
                    break;
```

```
                case "V":
```

```
                case "v":
```

```
                    Console.WriteLine("You wish to view the address list.");
```

```
                    break;
```

```
                case "Q":
```

```
                case "q":
```

```
                    Console.WriteLine("Bye.");
```

```
                    break;
```

```
                default:
```

```

        Console.WriteLine("{0} is not a valid choice", myChoice);
        break;
    }
    Console.Write("Press Enter key to continue...");
    Console.ReadLine();
    Console.WriteLine();
} while (myChoice != "Q" && myChoice != "q");
}
}

```

مثال 4-2 نحوه استفاده از حلقه **do-while** را نشان می‌دهد. ساختار نوشتاری این دستور بصورت زیر است :

```

do
{ <statements> } while (<boolean expression>);

```

دستورالعملهای مورد استفاده در بلوک این دستور، هر دستورالعمل معتبر زبان C# می‌تواند باشد. عبارت منطقی نیز همانند نمونه‌هایست که تا کنون با آنها آشنا شدیم و یکی از دو مقدار **true** یا **false** را بر می‌گرداند.

یکی از مصارف عمده حلقه **do** به جای حلقه **while**، مواردی است که می‌خواهیم یکسری دستورالعمل خاص، که آنها را درون بلوک **do** قرار می‌دهیم، حداقل یکبار اجرا شوند. در این مثال ابتدا یک منو برای کاربر نمایش داده می‌شود و سپس ورودی از وی دریافت می‌گردد. چون حلقه **while** عبارت منطقی خود در ابتدای اجرای حلقه بررسی می‌نماید، از اینرو تضمینی برای اجرای دستورات درون بلوک وجود نخواهد داشت، مگر شما بطور صریح برنامه را طوری طراحی نمایید که این عمل اتفاق بیفتد.

یک نگاه کلی به مثال 4-2 بیندازیم. در متد **Main()** متغیر **myChoice** را از نوع رشته‌ای تعریف نموده‌ایم. سپس یکسری دستورات را بر روی کنسول چاپ نموده‌ایم. این دستورات منوهای انتخاب برای کاربر هستند. ما باید ورودی از کاربر دریافت کنیم که چون این عمل از طریق **Console.ReadLine()** صورت گرفته، باید در متغیری از نوع رشته‌ای قرار گیرد و از اینرو این ورودی را در **myChoice** قرار داده‌ایم. ما باید ورودی را از کاربر دریافت کنیم و بر روی آن پردازش انجام دهیم. یک روش کارآمد برای این منظور استفاده از دستورالعمل **switch** است. همانطور که در دستور **switch** ملاحظه می‌کنید، برای **default** نیز دستوری در نظر گرفته شده است که نشان می‌دهد مقدار ورودی معتبر نیست.

حلقه **for**

به مثال 4-3 توجه کنید.

```

using System;

class ForLoop
{
    public static void Main()
    {
        for (int i=0; i < 20; i++)
        {

```

```

    if (i == 10)
        break;
    if (i % 2 == 0)
        continue;
    Console.WriteLine("{0} ", i);
}
Console.WriteLine();
}
}

```

مثال 4-3 یک حلقه **for** را نشان می‌دهد. استفاده از حلقه **for** برای زمانی مناسب است که دقیقاً بدانید که حلقه چندبار باید تکرار شود. محتویات درون پرانتزهای حلقه **for** از سه قسمت تشکیل شده است :

<initializer list>; <boolean expression>; <postloopaction list>

initializer list لیستی از عبارات است که بوسیله کاما از یکدیگر جدا می‌شوند. این عبارات تنها یکبار در طول دوره کاری حلقه **for** پردازش می‌شوند. همانطور که در مثال 4-3 نیز ملاحظه می‌کنید، این قسمت معمولاً برای تعیین متغیری عددی جهت آغاز عمل شمارش مورد استفاده قرار می‌گیرد.

پس از اینکه عبارتهای دورن **initializer list** پردازش شد، حلقه **for** به سراغ قسمت بعدی، یعنی عبارات منطقی (**boolean expression**) می‌رود. در این قسمت تنها یک عبارت منطقی می‌توان قرار داد ولی هر اندازه که بخواهید می‌توانید این عبارت منطقی را پیچیده نمایید، فقط توجه نمایید که این عبارت باید بگونه‌ای شود که مقدار **true** یا **false** برگرداند. از این عبارت منطقی معمولاً جهت کنترل متغیر شمارشی استفاده می‌شود.

هنگامیکه عبارت منطقی مقدار **true** بازگرداند، دستورالعمل‌های بلوک **for** اجرا می‌شوند. در مثال 4-3 ما از دو دستور **if** درون حلقه **for** نیز استفاده کرده‌ایم. اولین دستور **if** بررسی می‌کند که آیا مقدار متغیر **i** برابر با 10 هست یا نه. در اینجا یک نمونه دیگر از استفاده دستور **break** را ملاحظه می‌کنید. عملکرد دستور **break** در اینجا نیز همانند مورد استفاده آن در دستور **switch** است. در صورت اجرای دستور **break** اجرای حلقه **for** خاتمه یافته و اجرای برنامه به اولین دستور بعد از حلقه **for** منتقل می‌شود.

دومین دستور **if** با اسقاده از عملگر باقیمانده (%) بررسی می‌کند که آیا متغیر **i** بر 2 بخش پذیر هست یا نه. در صورتیکه متغیر **i** بر 2 بخش پذیر باشد، دستور **continue** اجرا می‌شود. پس از اجرای دستور **continue** از سایر دستورات حلقه **for** که بعد از **continue** قرار گرفته‌اند صرفه نظر می‌شود و اجرای برنامه به اول حلقه **for** باز می‌گردد. قسمت سوم در حلقه **for**، قسمت **postloopaction list** است. پس از اینکه تمامی دستورات درون حلقه **for** اجرا شد، اجرای حلقه به این قسمت باز می‌گردد. این قسمت لیستی از عملیاتی است که می‌خواهید پس از اجرای دستورات درون بلوک حلقه **for** انجام شوند. در مثال 4-3 این عمل، اضافه کردن یک واحد به متغیر شمارشی است. پس از افزوده شدن یک واحد به متغیر شمارشی، عبارت منطقی مجدداً مورد بررسی قرار می‌گیرد و در صورتیکه مقدار این عبارت برابر با **true** باشد، حلقه **for** مجدداً اجرا می‌گردد. حلقه **for** تا زمانی که عبارت منطقی برابر با **true** باشد اجرا می‌شود.


```
using System;
```

```
class ForEachLoop
```

```
{
    public static void Main()
    {
        string[] names = {"Meysam", "Ghazvini", "C#", "Persian"};
        foreach (string person in names)
        {
            Console.WriteLine("{0} ", person);
        }
    }
}
```

حلقه *foreach* برای پیمایش مجموعه‌ها بسیار مناسب است. یک نمونه از مجموعه‌ها در *C#*، آرایه‌ها هستند که در مثال 4-4 نیز مورد استفاده قرار گرفته است. اولین کاری که در متد *Main()* صورت گرفته، اعلان آرایه *names* از نوع رشته‌ای و مقدار دهی آن است.

درون پرانتزهای *foreach* عبارتی متشکل از دو المان قرار دارد که این المان‌ها بوسیله کلمه کلیدی *in* از یکدیگر جدا شده‌اند. المان سمت راست، مجموعه‌ایست که می‌خواهید اعضای آنرا مورد پیمایش قرار دهید. المان سمت چپ، متغیری از نوع مجموعه مورد نظر است که مقادیر پیمایش شده را بر می‌گرداند.

در هربار پیمایش، عنصری جدیدی از مجموعه درخواست می‌شود. این درخواستها از طریق متغیر قط خواندنی تعریف شده درون پرانتزهای *foreach* بازگردانده می‌شوند. تا زمانیکه عنصری در مجموعه وجود داشته باشد که مورد پیمایش قرار نگرفته است، حلقه *foreach* به کار خود ادامه خواهد داد زیرا عبارت منطقی حلقه *foreach* مقدار *true* را باز می‌گرداند. به محض اینکه تمامی عناصر مجموعه پیمایش شد، عبارت منطقی برابر با *false* شده و اجرای حلقه *foreach* خاتمه می‌یابد. در این حالت اجرای برنامه به اولین دستور بعد از حلقه *foreach* منتقل می‌گردد.

در مثال 4-4، متغیری از نوع رشته با نام *person* برای نگهداری عناصری که از آرایه *names* خوانده می‌شود، تعریف کرده‌ایم. تا زمانیکه اسمی در آرایه *names* وجود داشته باشد، در متغیر *person* قرار می‌گیرد و درون حلقه *foreach* بوسیله دستور *Console.WriteLine()* در خروجی نمایش داده می‌شود.

نکته: یکی از مهمترین ویژگیهای حلقه *foreach* در آنست که فقط می‌تواند عناصر یک مجموعه را بخواند و نمی‌تواند تغییری در آنها ایجاد نماید. مزیت دیگر آن، پیمایش تمامی عناصر مجموعه بدون اطلاع از تعداد عناصر موجود در آن است.

خلاصه

در این درس با نحوه کار با دستورالعمل‌های **for**، **while**، **do-while** و **foreach** آشنا شدید. حلقه **while** تا زمانی که شرطش صحیح باشد انجام می‌شود بدین معنی که تا زمانی که عبارت منطقی آن دارای مقدار **true** باشد، دستورات درون بلوک آن اجرا می‌شوند. حلقه **do**، دستورات بلوک خود را حداقل یکبار اجرا می‌کند و سپس شرط مورد نظر را بررسی می‌نماید و در صورتیکه عبارت منطقی آن مقدار **true** بازگرداند، دستورات بلوک خود را تکرار می‌نماید. حلقه **for** دستورات بلوک خود را به تعداد دفعات مشخص اجرا می‌نماید و حلقه **foreach** عناصر یک مجموعه را مورد پیمایش قرار می‌دهد. در نهایت نیز اشاره می‌شود که روند اجرای حلقه‌ها با استفاده از دستورات **break** و **continue** تغییر می‌نماید.

درس پنجم – متدها

در این قسمت با متدها در زبان **C#** آشنا می‌شوید. اهداف این درس به شرح زیر می‌باشد :

ü درک ساختار یک متد

ü درک تفاوت بین متدهای استاتیک (**static methods**) و متدهای نمونه (**instance**)

ü ایجاد نمونه جدید از اشیاء

ü نحوه فراخوانی متدها

ü درک چهار گونه متفاوت پارامترها

ü نحوه استفاده از مرجع **this**

تا کنون تمامی اعمالی که ما در برنامه‌هایمان انجام می‌دادیم در متد **Main()** اتفاق می‌افتادند. این روش برای برنامه‌های ساده و ابتدایی که استفاده کردیم مناسب بود، اما اگر برنامه‌ها پیچیده‌تر شوند و تعداد کارهای مورد نظر ما گسترش یابد، استفاده از متدها جایگزین روش قبل می‌گردد. متدها فوق‌العاده مفید هستند، چراکه کارها را به بخشهای کوچکتر و مجزا تقسیم می‌کنند و در نتیجه استفاده از آنها آسان‌تر خواهد بود. ساختار کلی یک متد به صورت زیر است :

[attributes][modifiers] return-type method-name ([parameters]) { statements }

دو قسمت **attributes** و **modifiers** را در آینده مورد بررسی قرار خواهیم داد. **return-type** نوعی است یک متد باز می‌گرداند و می‌تواند هر یک از انواع تعریف شده زبان **C#** و یا از انواع تعریف شده توسط کاربر باشد. هر متد با نام آن شناخته می‌شود. **method-name** نام انتخابی برنامه‌نویس برای یک متد است و از طریق همین نام فراخوانی متد انجام می‌شود. پارامترها (**parameters**) مولفه‌ها یا متغیرهایی هستند که برای انجام یکسری پردازش به متد ارسال می‌شوند و از طریق آنها می‌توان اطلاعاتی را به متد ارسال و یا از آن دریافت نمود، و در نهایت دستورالعملهای متد، دستورهایی از زبان **C#** هستند که بوسیله آنها عمل مورد نظر برنامه‌نویس انجام می‌شود و عملی است که یک متد آنرا انجام می‌دهد. مثال 1-5 پیاده‌سازی یک متد ساده را نمایش می‌دهد.

using System;

class OneMethod

{

public static void Main()

{

string myChoice;

OneMethod om = **new** OneMethod();

do

{

myChoice = om.getChoice();

// تصمیمی بر اساس انتخاب کاربر گرفته می شود

switch(myChoice)

{

case "A":

case "a":

Console.WriteLine("You wish to add an address.");

break;

case "D":

case "d":

Console.WriteLine("You wish to delete an address.");

break;

case "M":

case "m":

Console.WriteLine("You wish to modify an address.");

break;

case "V":

case "v":

Console.WriteLine("You wish to view the address list.");

break;

case "Q":

case "q":

Console.WriteLine("Bye.");

break;

default:

Console.WriteLine("{0} is not a valid choice", myChoice);

break;

}

// اجرای برنامه برای دیدن نتایج موقف می شود

Console.WriteLine();

Console.Write("Press Enter key to continue...");

Console.ReadLine();

Console.WriteLine();

} **while** (myChoice != "Q" && myChoice != "q");

// اجرای برنامه تا زمانیکه کاربر بخواهد ادامه می‌یابد

```
}  
string getChoice()  
{  
    string myChoice;  
    // منویی را نمایش می‌دهد  
    Console.WriteLine("My Address Book\n");  
    Console.WriteLine("A - Add New Address");  
    Console.WriteLine("D - Delete Address");  
    Console.WriteLine("M - Modify Address");  
    Console.WriteLine("V - View Addresses");  
    Console.WriteLine("Q - Quit\n");  
    Console.Write("Choice (A,D,M,V,or Q): ");  
    // ورودی دریافتی از کاربر را بررسی می‌کند  
    myChoice = Console.ReadLine();  
    Console.WriteLine();  
    return myChoice;  
}  
}
```

برنامه مثال 1-5 دقیقاً همان برنامه در س 4 است، با این تفاوت که در درس چهارم چاپ منو و دریافت ورودی از کاربر در متد **Main()** صورت می‌گرفت در حالیکه در این مثال، این اعمال در یک متد مجزا بنام **getChoice()** صورت می‌گیرد. نوع بازگشتی این متد از نوع رشته‌ای است. از این رشته در دستور **switch** در متد **Main()** استفاده می‌شود. همانطور که ملاحظه می‌نمایید، پراوتزهای متد **getChoice()** خالی هستند، یعنی این متد دارای پارامتر نیست، از اینرو هیچ اطلاعاتی به / از این متد منتقل نمی‌شود.

درون این متد، ابتدا متغیر **myChoice** را اعلان نموده‌ایم. هرچند نام و نوع این متغیر همانند متغیر **myChoice** موجود در متد **Main()** است، اما این دو متغیر دو متغیر کاملاً مجزا از یکدیگر می‌باشند. هر دو این متغیرها، متغیرهای محلی (**Local**) هستند، از اینرو تنها درون بلوکی که تعریف شده‌اند قابل دسترسی می‌باشند. به بیان دیگر این دو متغیر از وجود یکدیگر اطلاعی ندارند.

متد **getChoice()** منویی را در کنسول نمایش می‌دهد و ورودی انتخابی کاربر را دریافت می‌نماید. دستور **return** داده‌ها را از طریق متغیر **myChoice** به متد فراخواننده آن، یعنی **Main()**، باز می‌گرداند. توجه داشته باشید که، نوع متغیری که توسط دستور **return** باز گردانده می‌شود، باید دقیقاً همانند نوع بازگشتی متد باشد. در این مثال نوع بازگشتی، رشته است.

در **C#** دو گونه متد وجود دارد. یکی متدهای استاتیک (**Static**) و دیگری متدهای نمونه (**Instance**). متدهایی که در اعلان خود شامل کلمه کلیدی **static** هستند، از نوع استاتیک هستند، بدین معنا که هیچ نمونه‌ای از روی این متد قابل ایجاد نیست و این تنها همین نمونه موجود قابل استفاده است. از روی متدهای استاتیک نمی‌توان شیء (**Object**) ایجاد

کرد. در صورتیکه در اعلان متد از کلمه کلیدی **static** استفاده نشده باشد، متد بعنوان متد نمونه در نظر گرفته می‌شود، بدین معنا که از روی آن می‌توان نمونه ایجاد کرد و شیء تولید نمود. هر یک از اشیاء ایجاد شده از روی این متدها، تمامی عناصر آن متد را دارای می‌باشند.

در این مثال، چون **getChoice()** بصورت استاتیک اعلان نشده است، پس باید برای استفاده از آن شیء جدیدی تولید شود. تولید شیء جدید بوسیله **OneMethod om = new OneMethod()** صورت می‌پذیرد. در سمت چپ این اعلان، مرجع این شیء جدید، یعنی **om**، قرار دارد که از نوع **OneMethod** است. در اینجا توجه به یک نکته بسیار مهم است، **om** به خودی خود شیء نیست، بلکه می‌تواند مرجعی به شیء از نوع **OneMethod()** را در خود نگه‌دارد. در سمت راست این اعلان، تخصیص شیء جدیدی از نوع **OneMethod()** به متغیر **om** صورت گرفته است. کلمه کلیدی **new** عملگری است که شیء جدیدی را در **heap** ایجاد می‌نماید. اتفاقی که اینجا روی داده اینست که نمونه جدیدی از **OneMethod()** در **heap** تولید شده و سپس به مرجع **om** تخصیص داده می‌شود. حال که نمونه‌ای از متد **OneMethod()** را به **om** تخصیص داده‌ایم، از طریق **om** می‌توانیم با این متد کار نماییم.

متدها، فیلدها و سایر اعضای یک کلاس از طریق عملگر نقطه "." قابل دسترس هستند. هنگامیکه می‌خواهیم متد **getChoice()** را فراخوانی کنیم، بوسیله عملگر نقطه از طریق **om** به آن دسترسی پیدا می‌نماییم: **om.getChoice()**. برای نگهداری مقداری که **getChoice()** بر می‌گرداند، از عملگر "=" استفاده نموده‌ایم. رشته بازگشتی از متد **getChoice()** درون متغیر محلی **myChoice** متد **Main()** قرار می‌گیرد. از این قسمت، اجرای برنامه همانند قبل است.

پارامترهای متد

به مثال 2-5 توجه کنید.

using System;

class Address

```
{  
    public string name;  
    public string address;  
} //Address پایان کلاس
```

class MethodParams

```
{  
    public static void Main()  
    {  
        string myChoice;  
        MethodParams mp = new MethodParams();  
        do  
        {  
            // منویی نمایش داده شده و ورودی از کاربر دریافت می‌گردد  
        }  
    }  
}
```



```

myChoice = mp.getChoice();
// تصمیمی بر اساس ورودی کاربر گرفته می شود
mp.makeDecision(myChoice);
// جهت دیدن نتایج توسط کاربر، اجرای برنامه موقتا موقوف می گردد
Console.WriteLine("Press Enter key to continue...");
Console.ReadLine();
Console.WriteLine();
} while (myChoice != "Q" && myChoice != "q");
// اجرای حلقه تا زمانی که کاربر بخواهد ادامه پیدا می نماید

} //Main متد پایان

// نمایش منو و دریافت ورودی از کاربر
string getChoice()
{
    string myChoice;
    // نمایش منو

    Console.WriteLine("My Address Book\n");
    Console.WriteLine("A - Add New Address");
    Console.WriteLine("D - Delete Address");
    Console.WriteLine("M - Modify Address");
    Console.WriteLine("V - View Addresses");
    Console.WriteLine("Q - Quit\n");
    Console.WriteLine("Choice (A,D,M,V,or Q): ");
    // دریافت ورودی کاربر

    myChoice = Console.ReadLine();
    return myChoice;
} //getChoice() متد پایان

// تصمیم گیری
void makeDecision(string myChoice)
{
    Address addr = new Address();
    switch(myChoice)
    {
        case "A":
        case "a":
            addr.name = "Meysam";
            addr.address = "C# Persian";
            this.addAddress(ref addr);
            break;
        case "D":
        case "d":
            addr.name = "Ghazvini";

```

```

        this.deleteAddress(addr.name);
        break;
    case "M":
    case "m":
        addr.name = "CSharp";
        this.modifyAddress(out addr);
        Console.WriteLine("Name is now {0}.", addr.name);
        break;
    case "V":
    case "v":
        this.viewAddresses("Meysam", "Ghazvini", "C#", "Persian");
        break;
    case "Q":
    case "q":
        Console.WriteLine("Bye.");
        break;
    default:
        Console.WriteLine("{0} is not a valid choice", myChoice);
        break;
    }
}
// وارد کردن یک آدرس
void addAddress(ref Address addr)
{
    Console.WriteLine("Name: {0}, Address: {1} added.", addr.name,
addr.address);
}
// حذف یک آدرس
void deleteAddress(string name)
{
    Console.WriteLine("You wish to delete {0}'s address.", name);
}
// تغییر یک آدرس
void modifyAddress(out Address addr)
{
    //خطا رخ می دهد
    Console.WriteLine("Name: {0}.", addr.name);
    addr = new Address();
    addr.name = "Meysam";
    addr.address = "C# Persian";
}
// نمایش آدرس ها
void viewAddresses(params string[] names)
{

```

```
foreach (string name in names)
{
    Console.WriteLine("Name: {0}", name);
}
}
```

مثال 2-5، نمونه تغییر یافته مثال 1-5 است که در آن تمامی برنامه ماژولار شده و به متدهای مختلف تقسیم شده است. در زبان C# چهار گونه پارامتر وجود دارند: **ref**، **out**، **params** و **value**. بمنظور آشنایی با پارامترها، در مثال 2-5 کلاسی با نام **Address** با دو فیلد از نوع رشته تولید کرده‌ایم.

درون متد **Main()**، متد **getChoice()** را فراخوانی کرده‌ایم تا از کاربر ورودی دریافت کنیم و این ورودی در متغیر رشته‌ای **myChoice** قرار می‌گیرد. سپس متغیر **myChoice** را بعنوان آرگومان به متد **makeDecision()** ارسال نموده‌ایم. در اعلان **myDecision()**، همانطور که ملاحظه می‌نمایید، پارامتر این متد از نوع رشته و با نام **myChoice** تعریف شده است. توجه نمایید که این متغیر نیز محلی است و تنها درون متد **makeDecision()** قابل استفاده است. هرگاه در اعلان متد، برای پارامترهای آن هیچ **modifier** آورده نشود، این پارامتر بعنوان **value** در نظر گرفته می‌شود. در مورد پارامترهای مقداری (**value parameter**)، اصل مقدار متغیر یا پارامتر به پشته (**Stack**) کپی می‌شود. متغیرهایی که بصورت مقداری بعنوان پارامتر برای یک متد ارسال می‌شوند، همگی محلی بوده و تغییرات ایجاد شده بر روی آنها به هیچ وجه تغییری بر روی متغیر اصلی ایجاد نمی‌نماید.

دستور **switch** در متد **makeDecision()** برای هر **case** یک متد را فراخوانی می‌نماید. فراخوانی این متدها با آنچه در متد **Main()** دید مقداری متفاوت است. علاوه بر مرجع **mp**، در این فراخوانی‌ها از کلمه کلیدی **this** نیز استفاده شده است. کلمه کلیدی **this** ارجاعی به شیء فعلی دارد.

متد **addAddress()** پارامتری از نوع **ref** دارد. وجود چنین پارامتری بدین معناست که مرجعی از این پارامتر به متد ارسال می‌شود و این مرجع همچنان به شیء اصلی درون **heap** نیز اشاره دارد چراکه آدرس شیء مورد نظر به متد کپی می‌شود. در مورد پارامترهای **ref**، هرگونه تغییری که بر روی متغیر محلی رخ دهد، همان تغییر بر روی متغیر اصلی نیز اعمال می‌گردد. امکان تغییر مرجع وجود ندارد و تنها شیء ای که مورد آدرس‌دهی واقع شده، می‌تواند تغییر پیدا نماید. پارامترهای مرجعی (**ref**) را می‌توان به عنوان عناصر ورودی/خروجی برای متد در نظر گرفت.

پارامترهای **out** در مواردی استفاده می‌شوند که ارسال اطلاعات به متد از طریق پارامتر مد نظر نباشد، بلکه ارسال اطلاعات از متد مورد نظر باشد. استفاده از این پارامترها از اینرو کارآمد هستند که برنامه مجبور به کپی کردن پارامتر به متد نیست و از حجم سرباره (**Overhead**) برنامه می‌کاهد. در برنامه‌های عادی این مسئله چندان به چشم نمی‌آید، اما در برنامه‌های تحت شبکه که سرعت ارتباط و انتقال داده‌ها بسیار مهم است، این پارامترها ضروری می‌شوند.

متد **modifyAddress()** دارای پارامتری از نوع **out** است. پارامترهای **out** فقط به متد فراخواننده آن بازگشت داده می‌شوند. از آنجائیکه این پارامترها از متد فراخواننده هیچ مقداری دریافت نمی‌کنند و فقط درون متدی که به عنوان پارامتر به آن ارسال شده‌اند قابلیت تغییر دارند، از اینرو درون این متدهایی که به آنها ارسال می‌شوند، قبل از اینکه بتوان

از آنها استفاده نمود باید مقداری به آنها تخصیص داده شود. اولین خط در متد **modifyAddress()** بصورت توضیحات نوشته شده است. این خط را از حالت توضیحات خارج کرده و سپس برنامه اجرا کنید تا ببینید چه اتفاقی رخ خواهد داد. هنگامیکه این پارامتر مقدار دهی شود و مقداری را به متد فراخواننده خود بازگرداند، این مقدار بر روی متغیر متد فراخواننده کپی می‌گردد. توجه نمایید که پارامترهای **out** می‌بایست قبل از دستور **return** درون متد مقدار دهی شده باشند.

یکی از ویژگیهای مفید زبان **C#**، وجود پارامترهای **params** است که بوسیله آنها می‌توان متدی را اعلان کرد که تعداد متغیری متغیر را به عنوان پارامتر دریافت نماید. پارامترهای **params** حتماً باید یکی از انواع آرایه تک بعدی و یا آرایه دندانه‌دار (**Jagged Array**) باشند. در متد **makeDecision()** چهار متغیر رشته‌ای را به متد **viewAddresses()** ارسال نموده‌ایم که این متد پارامترهای خود را بصورت **params** دریافت می‌نماید. همانطور که ملاحظه می‌نمایید، تعداد متغیرهای ارسالی به متد می‌تواند متغیر باشد اما دقت داشته باشید که تمامی این متغیرها در یک آرایه تک بعدی قرار گرفته‌اند. درون متد **viewAddresses()** نیز با استفاده از دستور **foreach** تمامی عناصر موجود در این آرایه را نمایش داده‌ایم. پارامترهای **params** فقط متغیرهای ورودی دریافت می‌نمایند و تغییرات اعمال شده تنها بر روی متغیر محلی تاثیر می‌گذارد.

خلاصه

در این درس، با ساختار کلی یک متد آشنا شدید. فراگرفتید که در زبان **C#** چهار نوع پارامتر برای متدها وجود دارند. پارامترهای **value**، **ref**، **out** و **params**. همانطور که گفته شد حالت پیش فرض برای پارامترها، **value** است مگر آنکه صریحاً مشخص گردد.

درس ششم – Namespaces

اهداف این درس به شرح زیر می‌باشد:

ü آشنایی با **Namespace** در زبان **C#**

ü چگونگی استفاده از هدایتگر **using** (using directive)

ü چگونگی استفاده از هدایتگر **alias** (alias directive)

ü اعضای یک **Namespace** چه هستند؟

اگر به خاطر داشته باشید، در درس اول، در ابتدای برنامه از **using System;** استفاده نمودیم. با استفاده از این کد، امکان دسترسی ما به اعضای **Namespace** مورد نظر، که در اینجا **System** است، فراهم می‌شد. پس از مطالعه این درس، مطالب بسیاری درباره هدایتگر **using** فرا خواهید گرفت.

Namespace ها، المان‌های زبان **C#** هستند که شما را در سازماندهی کردن برنامه، کمک می‌کنند. پیاده‌سازی **Namespace** ها در کد برنامه بسیار مفید است چراکه از ایجاد مشکلات مربوط به استفاده مجدد کد، پیشگیری می‌نماید.

چگونگی ایجاد **Namespace**

به مثال زیر توجه نمایید.

// **Namespace** اعلان

using System;

// **C# Persian Namespace**

namespace csharp-persian

{

 // کلاس آغازین برنامه

class NamespaceCSS

 {

 // آغاز اجرای برنامه

public static void Main()

 {

 // چاپ خروجی در کنسول

 Console.WriteLine("This is the new C# Persian Namespace.");

 }

 }

}

مثال 1-6 چگونگی ایجاد یک **Namespace** را نشان می‌دهد. در این مثال ما با قرار دادن کلمه کلیدی **namespace** در جلوی **csharp-persian** یک **Namespace** جدید ایجاد نمودیم. مثال 2-6، **Namespace** های تودرتو را نشان می‌دهد.

// **Namespace** اعلان

using System;

// **C# Persian Namespace**

namespace csharp-persian

{

namespace tutorial

 {

 // کلاس آغازین برنامه

class NamespaceCSS

 {

 // آغاز اجرای برنامه

public static void Main()


```

{
    // چاپ خروجی در کنسول
    Console.WriteLine("This is the new C# Persian Namespace.");
}
}
}
}

```

Namespace ها این امکان را فراهم می آورند تا سیستمی جهت سازماندهی کدهای خود ایجاد نمایید. یک روش مناسب جهت سازماندهی **Namespace** های برنامه، استفاده از یک سیستم سلسله مراتبی است. بدین ترتیب که **Namespace** های عمومی را در بالای این ساختار قرار داده و هر چه در این ساختار پایین تر می آیم، **Namespace** های تخصصی تر قرار می گیرند. این سیستم سلسله مراتبی بوسیله **Namespace** های تودرتو قابل پیاده سازی هستند. این پیاده سازی در مثال 2-6 نشان داده شده است. با قرار دادن کدهای خود در **Namespace** های فرعی می توانید کدهای خود را سازماندهی نمایید.

// **Namespace** اعلان

using System;

// **C# Persian Tutorial Namespace**

namespace csharp-persian.tutorial

{

// کلاس آغازین برنامه

class NamespaceCSS

{

// آغاز اجرای برنامه

public static void Main()

{

// چاپ خروجی در کنسول

Console.WriteLine("This is the new C# Persian Namespace.");

}

}

}

مثال 3-6 روش دیگر ایجاد **Namespace** های تودرتو را نشان می دهد. همانطور که در این مثال مشاهده می نمایید، **Namespace** تودرتو با قرار دادن عملگر نقطه "." در بین *csharp-persian* و *tutorial* ایجاد شده است. این مثال از نظر کارایی دقیقاً همانند مثال 2-6 است و از نظر پیاده سازی آسان تر از آن می باشد.

فراخوانی اعضای یک *Namespace*

// **Namespace** اعلان

using System;

```

namespace csharp-persian
{
    // namespace تودرتو
    namespace tutorial
    {
        class myExample1
        {
            public static void myPrint1()
            {
                Console.WriteLine("First Example of calling another namespace
member.");
            }
        }
    }
    // کلاس آغازین برنامه
    class NamespaceCalling
    {
        // آغاز اجرای برنامه
        public static void Main()
        {
            // چاپ خروجی در کنسول
            tutorial.myExample1.myPrint1();
            tutorial.myExample2.myPrint2();
        }
    }
}

// Namespace مشابه Namespace تودرتو بالا
namespace csharp-persian.tutorial
{
    class myExample2
    {
        public static void myPrint2()
        {
            Console.WriteLine("Second Example of calling another namespace
member.");
        }
    }
}

```

مثال 4-6 چگونگی فراخوانی اعضای Namespace نشان می‌دهد. در ابتدای مثال 4-6، یک Namespace تودرتو اعلان شده است، *tutorial* درون *csharp-persian*، که دارای کلاس *myExample1* و متد *myPrint1* می‌باشد.

متد `Main()` این متد را با نام `tutorial.myExample1.myPrint1` فراخوانی می‌نماید. چون متد `Main()` و `tutorial` درون یک `Namespace` قرار دارند، دیگر نیازی به استفاده از نام `csharp-persian` غیر ضروری است. در انتهای مثال 4-6، یک `Namespace` دیگر بصورت `csharp-persian.tutorial` آورده شده است. کلاسهای `myExample1` و `myExample2` هر دو متعلق به یک `Namespace` می‌باشند، هرچند می‌تواند آنها را مجزا از یکدیگر نوشت در حالیکه متعلق به یک `Namespace` باشند. درون متد `Main()`، متد `myPrint2` بصورت `tutorial.myExample2.myPrint2` فراخوانی شده است. هرچند کلاس `myExample2` خارج از محدوده‌ایست که متد `myPrint2` فراخوانی شده است، نیازی به آمدن نام `csharp-persian` برای این فراخوانی وجود ندارد، زیرا هر دو کلاس متعلق به یک `Namespace`، یعنی `csharp-persian` هستند.

توجه نمایید که برای هر یک از کلاسهای `myExample1` و `myExample2` نامهای متفاوتی انتخاب شده است، زیرا هر عضو متعلق به یک `Namespace` باید دارای نام یکتایی باشد. برای متدهای `myPrint1` و `myPrint2`، بعلت سادگی در امر یادگیری این مطلب، نامهای متفاوتی در نظر گرفته شده است. توجه نمایید که این دو متد بعلت اینکه در کلاسهای متفاوتی قرار دارند، می‌توان نامهای یکسانی برایشان انتخاب نمود، و در اینصورت هیچگونه تداخلی بین ایندو ایجاد نخواهد شد.

استفاده از هدایتگر `using`

// **Namespace** اعلان

```
using System;  
using csharp_station.tutorial;
```

// کلاس آغازین برنامه

```
class UsingDirective
```

```
{
```

// آغاز اجرای برنامه

```
public static void Main()
```

```
{
```

// **Namespace** از عضوی

```
myExample.myPrint();
```

```
}
```

```
}
```

// **C# PersianTutorial Namespace**

```
namespace csharp-persian.tutorial
```

```
{
```

```
class myExample
```

```
{
```

```
public static void myPrint()
```

```
{
```

```
Console.WriteLine("Example of using a using directive.");
```

```
}
```

```
}
```

```
}  
}
```

در صورتیکه می خواهید متدهایی را بدون استفاده از نام کامل آنها، به همراه نام **Namespace**، استفاده نمایید، می توانید از هدایتگر **using** استفاده نمایید. در مثال 5-6، دوبار از هدایتگر **using** استفاده شده است. اولین استفاده از این هدایتگر، **using System**، دقیقاً مشابه به همان چیزی است که تا کنون در این درسها مشاهده نموده اید. با استفاده از این هدایتگر می توانید از اعضای موجود در **System**، بدون اینکه نیاز به تایپ کلمه **System** در هر بار داشته باشید، استفاده نمایید. در **myPrint()** کلاس **Console** عضوی از **System** است که دارای متد **WriteLine()** می باشد. اگر در ابتدای برنامه از هدایتگر **using** استفاده نکنیم، باید برای هر دفعه استفاده از متد **WriteLine()**، نام کامل آن یعنی **System.Console.WriteLine()** را تایپ نماییم.

به طور مشابه، استفاده از **using cssharp-persian.tutorial** امکان استفاده از اعضای این **Namespace** را، بدون نیاز به تایپ نام کامل آنها فراهم می نماید. چون در ابتدای برنامه از هدایتگر **using** استفاده نموده ایم، در متن برنامه متد **myPrint()** را بصورت **myExample.myPrint()** استفاده نموده ایم، در صورتیکه از هدایتگر **using** استفاده نمی گردیم برای استفاده از این متد باید آنرا بصورت **cssharp-persian.tutorial.myExample.myPrint()** می آوردیم.

استفاده از هدایتگر **Alias**

// **Namespace** اعلان

```
using System;  
using csTut = cssharp-persian.tutorial.myExample; // alias
```

// کلاس آغازین برنامه

```
class AliasDirective
```

```
{
```

// آغاز اجرای برنامه

```
public static void Main()
```

```
{
```

// **Namespace** فراخوانی عضوی از

```
csTut.myPrint();
```

```
myPrint();
```

```
}
```

// متدی که ممکن است تولید ابهام نماید.

```
static void myPrint()
```

```
{
```

```
Console.WriteLine("Not a member of cssharp-  
persian.tutorial.myExample.");
```

```
}
```

```

}
// C# Persian Tutorial Namespace
namespace csharp-persian.tutorial
{
    class myExample
    {
        public static void myPrint()
        {
            Console.WriteLine("This is a member of csharp-
persian.tutorial.myExample.");
        }
    }
}

```

در برخی موارد ممکن است با **Namespace** خاصی روبرو شوید که دارای نامی طولانی است و تمایل داشته باشید تا نام آنرا کوتاهتر نمایید. این کار با استفاده از ایجاد استعاره (**Alias**) امکان‌پذیر است. همانطور که در مثال 6-6 ملاحظه می‌نمایید، با استفاده از

`csTut = csharp-persian.tutorial.myExample` یک استعاره تولید کرده‌ایم و در متن برنامه به جای استفاده از نام طولانی `csharp-persian.tutorial.myExample` از نام مستعار آن یعنی `csTut` استفاده نموده‌ایم. حال از این نام مستعار در هر جای برنامه می‌توان استفاده کرد. ما در این برنامه در متد `Main()` استفاده نموده‌ایم.

همچنین در متد `Main()` فراخوانی از متد `myPrint()` مربوط به کلاس `AliasDirective` صورت گرفته است. نام این متد همانند `myPrint()` موجود در `myExample` است. علت اینکه می‌توان هر دو این متدها را همزمان فراخوانی کرد، استفاده از هدایتگر `alias` برای متد `myPrint()` مربوط به کلاس `myExample` است. (`csTut.myPrint()`) استفاده از این روش باعث می‌شود تا کامپایلر دریابد که کدام متد را باید اجرا نماید. در صورتیکه به اشتباه از هدایتگر `alias` (`csTut.myPrint()`) استفاده نکنیم، کامپایلر به اشتباه متد `myPrint()` مربوط به کلاس `AliasDirective` را دوبار اجرا خواهد کرد.

تا اینجا، ما تنها کلاسها را در **Namespace** ها نشان دادیم. **Namespace** ها انواع دیگری را نیز می‌توانند در خود داشته باشند که در زیر مشاهده می‌نمایید :

Classes
Structures
Interfaces
Enumerations
Delegates

خلاصه

در این درس با **Namespace** آشنا شدید و فراگرفتید که چگونه **Namespace** خود را اعلان نمایید. اگر نخواهید در هربار استفاده از متدها یا اعضای یک **Namespace**، نام کامل آنها را استفاده کنید، باید از هدایتگر `using` استفاده نمایید. در صورتیکه بخواهید بجای استفاده از نام طولانی یک **Namespace**، از نامی کوتاه‌تر استفاده کنید، باید از هدایتگر استعاره‌ای (**Alias Directive**) استفاده نمایید.

درس هفتم – آشنایی با کلاسها در C#

در این درس با کلاسها در زبان C# آشنا خواهید شد. اهداف این درس به شرح زیر می‌باشند :

- ü پیاده‌سازی سازنده‌ها (Constructors)
- ü درک تفاوت بین اعضای نمونه (Instance) و استاتیک (Static)
- ü آشنایی با تخریب‌کننده‌ها (Destructors)
- ü آشنایی با اعضای کلاسها

در تمامی مطالبی که در این سایت مشاهده کرده‌اید، برنامه‌ها دارای کلاس‌هایی بوده‌اند. در حال حاضر باید درک نسبی از کلاسها و کار آنها و چگونگی ایجاد آنها داشته باشید. در این درس مروری بر آموخته‌های قبلی از کلاسها خواهیم کرد و نیز با اعضای کلاسها آشنا می‌شویم.

یک کلاس با استفاده از کلمه کلیدی **class** که بدنبال آن نام کلاس آمده باشد، اعلان می‌گردد و اعضای این کلاس درون {} اعلان می‌گردند. هر کلاس دارای سازنده‌ای می‌باشد که در هربار ایجاد نمونه‌ای جدید از آن کلاس، بصورت خودکار فراخوانی می‌گردد. هدف از سازنده، تخصیص‌دهی اعضای کلاس در زمان ایجاد نمونه‌ای جدید از کلاس است. سازنده‌ها دارای مقادیر بازگشتی نبوده و همواره نامی مشابه نام کلاس دارند. مثال 1-7 نمونه‌ای از یک کلاس را نشان می‌دهد.

// Namespace اعلان

using System;

class OutputClass

{

string myString;

// سازنده

public OutputClass(string inputString)

{

myString = inputString;

}

// متد نمونه

public void printString()

{

Console.WriteLine("{0}", myString);

}

// تخریب‌کننده

~OutputClass()

{

// روتینی جهت آزادسازی برخی از منابع سیستم

}

```

}
// کلاس آغازین برنامه
class ExampleClass
{
    // آغاز اجرای برنامه
    public static void Main()
    {
        // نمونه‌ای از OutputClass
        OutputClass outCl = new OutputClass("This is printed by the output
class.");
        // فراخوانی متد کلاس Output
        outCl.printString();
    }
}

```

در مثال 1-7 دو کلاس دیده می‌شوند. کلاس بالایی، کلاس *OutPutClass*، دارای سازنده، متد نمونه و یک تخریب کننده است. همچنین این کلاس دارای فیلدی با نام *myString* است. توجه نمایید که چگونه سازنده این کلاس اعضای آنرا تخصیص‌دهی (مقداردهی) می‌نماید. در این مثال، سازنده کلاس رشته ورودی (*inputString*) را بعنوان آرگومان خود دریافت می‌نماید. سپس این مقدار داخل فیلد کلاس یعنی *myString* کپی می‌گردد.

همانطور که در *ExampleClass* مشاهده می‌نمایید، استفاده از سازنده الزامی نمی‌باشد. در این مورد سازنده پیش فرض ایجاد می‌گردد. سازنده پیش فرض، سازنده‌ای بدون هیچ نوع آرگومانی است. البته شایان ذکر است که سازنده‌هایی بدون آرگومان همیشه مورد استفاده نبوده و مفید نیستند. جهت کارآمد کردن بیشتر سازنده‌های بدون آرگومان بهتر است آنها را با تخصیص‌دهنده (*Initializers*) پیاده‌سازی نمایید. به مثال زیر در این زمینه توجه نمایید :

```

public OutputClass() : this("Default Constructor String") { }

```

فرض کنید این عبارت در کلاس *OutPutClass* در مثال 1-7 قرار داشت. این سازنده پیش فرض به یک تخصیص‌دهنده همراه شده است. ":" ابتدای تخصیص‌دهنده را مشخص می‌نماید، و به دنبال آن کلمه کلیدی *this* آمده است. کلمه کلیدی *this* به شیء کنونی اشاره می‌نماید. استفاده از این کلمه، فراخوانی به سازنده شیء کنونی که در آن تعریف شده است، ایجاد می‌کند. بعد از کلمه کلیدی *this* لیست پارامترها قرار می‌گیرد که در اینجا یک رشته است. عملی که تخصیص‌دهنده فوق انجام می‌دهد، باعث می‌شود تا سازنده *OutPutClass* رشته‌ای را بعنوان آرگومان دریافت نماید. استفاده از تخصیص‌دهنده‌ها تضمین می‌نمایند که فیلدهای کلاس شما در هنگام ایجاد نمونه‌ای جدید مقداردهی می‌شوند.

مثال فوق نشان داد که چگونه یک کلاس می‌تواند سازنده‌های متفاوتی داشته باشد. سازنده‌ای که فراخوانی می‌شود، به تعداد و نوع آرگومانهایش وابسته است.

در زبان *C#*، اعضای کلاسها دو نوع می‌باشند: اعضای نمونه و استاتیک. اعضای نمونه کلاس متعلق به رخداد خاصی از کلاس هستند. هربار که شیء از کلاسی خاص ایجاد می‌کنید، در حقیقت نمونه جدیدی از آن کلاس ایجاد کرده‌اید. متد

Main() در کلاس **ExampleClass** نمونه جدیدی از **OutPutClass** را تحت نام **outCl** ایجاد می‌نماید. می‌توان نمونه‌های متفاوتی از کلاس **OutPutClass** را با نامهای مختلف ایجاد نمود. هر یک از این نمونه‌های مجزا بوده و به تنهایی عمل می‌کنند. به عنوان مثال اگر دو نمونه از کلاس **OutPutClass** همانند زیر ایجاد نمایید :

```
OutputClass oc1 = new OutputClass("OutputClass1");  
OutputClass oc2 = new OutputClass("OutputClass2");
```

با این اعلان، شما دو نمونه از کلاس **OutPutClass** را ایجاد کرده‌اید که یک از آنها دارای فیلد **myString** و متد **printString()** هستند و این فیلدها و متدها کاملاً از یکدیگر مجزا می‌باشند. به بیان دیگر در صورتیکه عضوی از کلاس استاتیک باشد از طریق ساختار نوشتاری **<static class member>.<class name>** قابل دسترس خواهد بود. در این مثال نمونه‌ها **oc1** و **oc2** هستند. فرض کنید کلاس **OutPutClass** دارای متد استاتیک زیر باشد :

```
public static void staticPrinter()  
{  
    Console.WriteLine("There is only one of me.");  
}
```

این متد را از درون متد **Main()** به صورت زیر می‌توانید فراخوانی نمایید :

```
OutputClass.staticPrinter();
```

اعضای استاتیک یک کلاس تنها از طریق نام آن کلاس قابل دسترس می‌باشند و نه از طریق نام نمونه ایجاد شده از روی کلاس. بدین ترتیب برای فراخوانی اعضای استاتیک یک کلاس نیازی به ایجاد نمونه از روی آن کلاس نمی‌باشد. همچنین تنها یک کپی از عضو استاتیک کلاس، در طول برنامه موجود می‌باشد. یک مورد استفاده مناسب از اعضای استاتیک در مواردی است که تنها یک عمل باید انجام گیرد و در انجام این عمل هیچ حالت میانی وجود نداشته باشد، مانند محاسبات ریاضی. در حقیقت، **Net Framework BCL**، خود دارای کلاس **Math** می‌باشد که از اعضای استاتیک بهره می‌برد.

نوع دیگر سازنده‌ها، سازنده‌های استاتیک هستند. از سازنده‌های استاتیک جهت مقداره‌ی فیلدهای استاتیک یک کلاس استفاده می‌شود. برای اعلان یک سازنده استاتیک تنها کفایت که از کلمه کلیدی **static** در جلوی نام سازنده استفاده نمایید. سازنده استاتیک قبل از ایجاد نمونه جدیدی از کلاس، قبل از فراخوانی عضو استاتیک و قبل از فراخوانی سازنده استاتیک کلاس مشتق شده، فراخوانی می‌گردد. این سازنده‌ها تنها یکبار فراخوانی می‌شوند.

OutPutClass همچنین دارای یک تخریب‌کننده (**Destructor**) است. تخریب‌کننده‌ها شبیه به سازنده‌ها هستند، با این تفاوت که در جلوی خود علامت **"~"** را دارا می‌باشند. هیچ پارامتری دریافت نکرده و هیچ مقداری باز نمی‌گردانند. از تخریب‌کننده‌ها می‌توان در هر نقطه از برنامه که نیاز به آزادسازی منابع سیستم که در اختیار کلاس یا برنامه است، استفاده نمود. تخریب‌کننده‌ها معمولاً زمانی فراخوانی می‌شوند که **Garbage Collector** زبان **C#** تصمیم به حذف شیء مورد استفاده برنامه از حافظه و آزادسازی منابع سیستم، گرفته باشد. (**Garbage Collector** یا **GC**، یکی از امکانات **Net Framework**. مخصوص زبان **C#** است که سیستم بصورت اتوماتیک اقدام به آزادسازی حافظه و باز گرداندن منابع بلا استفاده به سیستم می‌نماید. فراخوانی **GC** بصورت خودکار رخ می‌دهد مگر برنامه‌نویس بصورت

صریح از طریق تخریب‌کننده‌ها آنرا فراخوانی نماید. در مباحث پیشرفته‌تری که در آینده مطرح می‌کنیم خواهید دید که در چه مواقعی نیاز به فراخوانی تخریب‌کننده‌ها بصورت شخصی دارید.)

تا کنون، تنها اعضای کلاس که با آنها سر و کار داشته‌اید، متدها، فیلدها، سازنده‌ها و تخریب‌کننده‌ها بوده‌اند در زیر لیست کاملی از انواعی را که می‌توانید در کلاس از آنها استفاده نمایید آورده شده است :

- Constructors
- Destructors
- Fields
- Methods
- Properties
- Indexers
- Delegates
- Events
- Nested Classes

مواردی که در این درس با آنها آشنا نشدید، حتماً در درس‌های آینده مورد بررسی قرار خواهند گرفت.

خلاصه

در این درس نحوه اعلان سازنده‌های استاتیک و نمونه را فرا گرفتید و با نحوه مقداره‌ی به فیلدها آشنا شدید. زمانیکه نیاز به ایجاد نمونه از روی شیء نباشد از اعضای استاتیک کلاس استفاده می‌کنیم. با استفاده از تخریب‌کننده‌ها می‌توانید منابع بلا استفاده را به سیستم باز گردانید.

درس هشتم – ارث‌بری کلاس‌ها

در این درس درباره ارث‌بری در زبان برنامه‌نویسی C# صحبت خواهیم کرد. اهداف این درس بشرح زیر می‌باشند :

ü پیاده‌سازی کلاسهای پایه (Base Class)

ü پیاده‌سازی کلاسهای مشتق شده (Derived Class)

ü مقدار دهی کلاس پایه از طریق کلاس مشتق شده

ü فراخوانی اعضای کلاس پایه

ü پنهان‌سازی اعضای کلاس پایه

ارث‌بری یکی از مفاهیم اساسی و پایه‌شی‌گرایی است. با استفاده از این ویژگی امکان استفاده مجدد از کد موجود فراهم می‌شود. بوسیله استفاده موثر از این ویژگی کار برنامه‌نویسی آسان‌تر می‌گردد.

ارث‌بری (Inheritance)

using System;

```
public class ParentClass
{
```

```

public ParentClass()
{
    Console.WriteLine("Parent Constructor.");
}
public void print()
{
    Console.WriteLine("I'm a Parent Class.");
}
}
public class ChildClass : ParentClass
{
    public ChildClass()
    {
        Console.WriteLine("Child Constructor.");
    }
    public static void Main()
    {
        ChildClass child = new ChildClass();
        child.print();
    }
}

```

خروجی این برنامه بصورت زیر است :

Parent Constructor.

Child Constructor.

I'm a Parent Class.

در مثال 1-8 دو کلاس وجود دارد. کلاس بالای *ParentClass* و کلاس پائینی *ChildClass* است. کاری که می‌خواهیم در اینجا انجام دهیم اینست که زیر کلاسی ایجاد کنیم که با استفاده از کدهای موجود در *ParentClass* عمل نماید.

برای این منظور ابتدا باید در اعلان *ChildClass* مشخص کنیم که این کلاس می‌خواهد از کلاس *ParentClass* ارث‌بری داشته باشد. این عمل با اعلان *public class ChildClass : ParentClass* روی می‌دهد. کلاس پایه با قرار دادن ":" بعد از نام کلاس مشتق شده معین می‌شود.

C# فقط از ارث‌بری یگانه پشتیبانی می‌نماید. از اینرو تنها یک کلاس پایه برای ارث‌بری می‌توان معین نمود. البته باید اشاره کرد که ارث‌بری چندگانه تنها از واسطه‌ها (**Interfaces**) امکان‌پذیر است که در درس‌های آینده به آنها اشاره می‌نماییم.

ChildClass دقیقاً توانایی‌های *ParentClass* را دارد. از اینرو می‌توان گفت *ChildClass* یک *ParentClass* است. (*ChildClass IS a ParentClass*) دارای متد *Print()* مربوط به خود نیست و از متد کلاس *ParentClass* استفاده می‌کند. نتیجه این عمل در خط سوم خروجی دیده می‌شود.

کلاسهای پایه به طور خودکار، قبل از کلاس‌های مشتق شده نمونه‌ای از روی آنها ایجاد می‌گردد. به خروجی مثال 8-1 توجه نمایید. سازنده *ParentClass* قبل از سازنده *ChildClass* اجرا می‌گردد.

برقراری ارتباط کلاس مشتق شده با کلاس پایه
به مثال 8-2 که در زیر آمده است توجه نمایید.

using System;

```
public class Parent
{
    string parentString;
    public Parent()
    {
        Console.WriteLine("Parent Constructor.");
    }
    public Parent(string myString)
    {
        parentString = myString;
        Console.WriteLine(parentString);
    }
    public void print()
    {
        Console.WriteLine("I'm a Parent Class.");
    }
}
public class Child : Parent
{
    public Child() : base("From Derived")
    {
        Console.WriteLine("Child Constructor.");
    }
    public new void print()
    {
        base.print();
        Console.WriteLine("I'm a Child Class.");
    }
    public static void Main()
    {
        Child child = new Child();
        child.print();
        ((Parent)child).print();
    }
}
```

خروجی این برنامه بشکل زیر است :

From Derived
Child Constructor.
I'm a Parent Class.
I'm a Child Class.
I'm a Parent Class.

کلاسهای مشتق شده در طول ایجاد نمونه می توانند با کلاس پایه خود ارتباط برقرار نمایند. در مثال 2-8 چگونگی انجام این عمل را در سازنده **ChildClass** نشان می دهد. استفاده از " : " و کلمه کلیدی **base** باعث فراخوانی سازنده کلاس پایه به همراه لیست پارامترهایش می شود. اولین سطر خروجی، فراخوانی سازنده کلاس پایه را به همراه رشته **"From Derived"** نشان می دهد.

ممکن است حالتی رخ دهد که نیاز داشته باشید تا متد موجود در کلاس پایه را خود پیاده سازی نمایید. کلاس **Child** این عمل را با اعلان متد **Print()** مربوط به خود انجام می دهد. متد **Print()** مربوط به کلاس **Child**، متد **Print()** کلاس **Parent** را پنهان می کند. نتیجه این کار آنست که متد **Print()** کلاس **Parent** تا زمانی که عمل خاصی انجام ندهیم قابل فراخوانی نمی باشد.

درون متد **Print()** کلاس **Child**، صریحاً متد **Print()** کلاس **Parent** را فراخوانی کرده ایم. این عمل با استفاده از کلمه کلیدی **base** قبل از نام متد انجام گرفته است. با استفاده از کلمه کلیدی **base** می توان به هر یک از اعضای **public** و **protected** کلاس پایه دسترسی داشت. خروجی مربوط به متد **Print()** کلاس **Child** در سطرها سوم و چهارم خروجی دیده می شوند.

روش دیگر دسترسی به اعضای کلاس پایه، استفاده از **Casting** صریح است. این عمل در آخرین سطر از متد **Main()** کلاس **Child** رخ داده است. توجه داشته باشید که کلاس مشتق شده نوع خاصی از کلاس پایه اش می باشد. این مسئله باعث می شود تا بتوان کلاس مشتق شده را مورد عمل **Casting** قرار داد و آنرا نمونه ای از کلاس پایه اش قرار داد. آخرین خط خروجی نشان می دهد که متد **Print()** کلاس **Parent** اجرا شده است.

به وجود کلمه کلیدی **new** در متد **Print()** کلاس **Child** توجه نمایید. این عمل باعث می شود تا متد **Print()** کلاس **Child** متد **Print()** کلاس پایه اش را پنهان نماید. در صورتیکه از کلمه کلیدی **new** استفاده نشود، کامپایلر پیغام خطاری را تولید می کند تا توجه شما را به این مسئله جلب کند. توضیحات بیشتر در این زمینه مربوط به مبحث چندریختی (**Polymorphism**) است که در درس آینده آنرا بررسی خواهیم نمود.

خلاصه

در این درس با روش ایجاد کلاس پایه و کلاس مشتق شده از آن آشنا شدید. متدهای کلای پایه را می توانید بصورت صریح و یا ضمنی فراخوانی کنید. همچنین متوجه شدید که کلاس مشتق شده نوع خاصی از کلاس پایه است.

درس نهم – چند ریختی (**Polymorphism**)

در این درس به بررسی چند ریختی در زبان **C#** خواهیم پرداخت. اهداف این درس عبارتند از :

- چند ریختی چیست؟
- پیاده‌سازی متد مجازی (Virtual Method)
- Override کردن متد مجازی
- استفاده از چند ریختی در برنامه‌ها

یکی دیگر از مفاهیم پایه‌ای در شی گرای، چند ریختی (Polymorphism) است. با استفاده از این ویژگی، می‌توان برای متد کلاس مشتق شده پیاده‌سازی متفاوتی از پیاده‌سازی متد کلاس پایه ایجاد نمود. این ویژگی در جایی مناسب است که می‌خواهید گروهی از اشیاء را به یک آرایه تخصیص دهید و سپس از متد هر یک از آنها را استفاده کنید. این اشیاء الزاما نباید از یک نوع شیء باشند. هرچند اگر این اشیاء بواسطه ارث‌بری به یکدیگر مرتبط باشند، می‌توان آنها را بعنوان انواع ارث‌بری شده به آرایه اضافه نمود. اگر هر یک از این اشیاء دارای متدی با نام مشترک باشند، آنگاه می‌توان هر یک از آنها را جداگانه پیاده‌سازی و استفاده نمود. در این درس با چگونگی انجام این عمل آشنا می‌گردید.

متد مجازی (Virtual Method)

using System;

public class DrawingObject

```
{
    public virtual void Draw()
    {
        Console.WriteLine("I'm just a generic drawing object.");
    }
}
```

مثال 9-1 کلاس *DrawingObject* را نشان می‌دهد. این کلاس می‌تواند بعنوان کلاسی پایه جهت کلاسهای دیگر در نظر گرفته شود. این کلاس تنها دارای یک متد با نام *Draw()* می‌باشد. این متد دارای پیشوند *virtual* است. وجود کلمه *virtual* بیان می‌دارد که کلاسهای مشتق شده از این کلاس می‌توانند، این متد را *override* نمایند و آنرا به طریقه دلخواه پیاده‌سازی کنند.

using System;

public class Line : DrawingObject

```
{
    public override void Draw()
    {
        Console.WriteLine("I'm a Line.");
    }
}
```

public class Circle : DrawingObject

```
{
    public override void Draw()
```

```

    {
        Console.WriteLine("I'm a Circle.");
    }
}
public class Square : DrawingObject
{
    public override void Draw()
    {
        Console.WriteLine("I'm a Square.");
    }
}

```

در مثال [9-2](#)، سه کلاس دیده می‌شود. این کلاسها از کلاس *DrawingObject* ارث‌بری می‌کنند. هر یک از این کلاسها دارای متد *Draw()* هستند و تمامی آنها دارای پیشوند *override* می‌باشند. وجود کلمه کلیدی *override* قبل از نام متد، این امکان را فراهم می‌نماید تا کلاس، متد کلاس پایه خود را *override* کرده و آنرا به طرز دلخواه پیاده‌سازی نماید. متدهای *override* شده باید دارای نوع و پارامترهای مشابه متد کلاس پایه باشند.

پیاده‌سازی چند ریختی

```

using System;

public class DrawDemo
{
    public static int Main( )
    {
        DrawingObject[] dObj = new DrawingObject[4];
        dObj[0] = new Line();
        dObj[1] = new Circle();
        dObj[2] = new Square();
        dObj[3] = new DrawingObject();
        foreach (DrawingObject drawObj in dObj)
        {
            drawObj.Draw();
        }
        return 0;
    }
}

```

مثال [9-3](#) برنامه‌ای را نشان می‌دهد که از کلاسهای مثال [9-1](#) و [9-2](#) استفاده می‌کند. در این برنامه چند ریختی پیاده‌سازی شده است. در متد *Main()* یک آرایه ایجاد شده است. عناصر این آرایه از نوع *DrawingObject* تعریف شده است. این آرایه *dObj* نامگذاری شده و چهار عضو از نوع *DrawingObject* را در خود نگه می‌دارد.

سپس آرایه **dObj** تخصیص‌دهی شده است. به دلیل رابطه ارث‌بری این عناصر با کلاس **DrawingObject** عناصر **Circle**، **Line** و **Square** قابل تخصیص به این آرایه می‌باشند. بدون استفاده از این قابلیت، قابلیت ارث‌بری، برای هر یک از این عناصر باید آرایه‌ای جدا می‌ساختید. ارث‌بری باعث می‌شود تا کلاسهای مشتق شده بتوانند همانند کلاس پایه خود عمل کنند که این قابلیت باعث صرفه‌جویی در وقت و هزینه تولید برنامه می‌گردد.

پس از تخصیص‌دهی آرایه، حلقه **foreach** تک تک عناصر آنرا پیمایش می‌کند. درون حلقه **foreach** متد **Draw()** برای هر یک از اعضای آرایه اجرا می‌شود. نوع شیء مرجع آرایه **dObj** **DrawingObject** است. چون متد **Draw()** در هر یک از این اشیاء **override** می‌شوند، از اینرو متد **Draw()** مربوط به هر یک از این اشیاء اجرا می‌شوند. خروجی این برنامه بصورت زیر است :

I'm a Line.

I'm a Circle.

I'm a Square.

I'm just a generic drawing object.

متد **override** شده **Draw()** مربوط به هر یک از کلاسهای مشتق شده در برنامه فوق همانند خروجی اجرا می‌شوند. آخرین ط خروجی نیز مربوط به کلاس مجازی **Draw()** از کلاس **DrawingObject** است، زیرا آخرین عنصر آرایه شیء **DrawingObject** است.

خلاصه

در این درس با مفهوم کلی چند ریختی آشنا شدید. چند ریختی امکانی است که مخصوص زبانهای برنامه‌نویسی شی‌گرا است و از طریق آن می‌توان برای یک متد موجود در کلاس پایه، چندین پیاده‌سازی متفاوت در کلاسهای مشتق شده داشت.

درس دهم – ویژگیها در C#

در این درس با ویژگیها (**Properties**) در زبان C# آشنا خواهیم شد. اهداف این درس به شرح زیر می‌باشد :

- موارد استفاده از **Property** ها
- پیاده‌سازی **Property**
- ایجاد **Property** فقط خواندنی (**Read-Only**)
- ایجاد **Property** فقط نوشتنی (**Write-Only**)

Property ها امکان ایجاد حفاظت از فیلدهای یک کلاس را از طریق خواندن و نوشتن بوسیله **Property** را فراهم می‌نماید. **Property** ها علاوه بر اینکه از فیلدهای یک کلاس حفاظت می‌کنند، همانند یک فیلد قابل دسترسی هستند. بمنظور درک ارزش **Property** ها بهتر است ابتدا به روش کلاسیک کپسوله کردن متدها توجه نمایید.

مثال 10-1: یک نمونه از چگونگی دسترسی به فیلدهای کلاس به طریقۀ کلاسیک

```
using System;
public class PropertyHolder
{
    private int someProperty = 0;
    public int getSomeProperty()
    {
        return someProperty;
    }
    public void setSomeProperty(int propValue)
    {
        someProperty = propValue;
    }
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.setSomeProperty(5);
        Console.WriteLine("Property Value: {0}",
propHold.getSomeProperty());
        return 0;
    }
}
```

مثال 10-1 روش کلاسیک دسترسی به فیلدهای یک کلاس را نشان می‌دهد. کلاس **PropertyHolder** دارای فیلدی است تمایل داریم به آن دسترسی داشته باشیم. این کلاس دارای دو متد **getSomeProperty()** و **setSomeProperty()** می‌باشد. متد **getSomeProperty()** مقدار فیلد **someProperty** را باز می‌گرداند و متد **setSomeProperty()** مقداری را به فیلد **someProperty** تخصیص می‌دهد.

کلاس **PropertyTester** از متدهای کلاس **PropertyHolder** جهت دریافت مقدار فیلد **someProperty** از کلاس **PropertyHolder** استفاده می‌کند. در متد **Main()** نمونه جدیدی از شی **PropertyHolder** با نام **propHold** ایجاد می‌گردد. سپس بوسیله متد **setSomeProperty** مقدار **someMethod** از **propHold** برابر با 5 می‌گردد و سپس برنامه مقدار **property** را با استفاده از فراخوانی متد **Console.WriteLine()** در خروجی نمایش می‌دهد. آرگومان مورد استفاده برای بدست آوردن مقدار **property** فراخوانی به متد **getSomeProperty()** است که توسط آن عبارت **"Property Value : 5"** در خروجی نمایش داده می‌شود.

چنین متد دسترسی به اطلاعات فیلد بسیار خوب است چرا که از نظریه کپسوله کردن شیءگرایی پشتیبانی می‌کند. اگر پیاده‌سازی **someProperty** نیز تغییر یابد و مثلاً از حالت **int** به **byte** تغییر یابد، باز هم این متد کار خواهد کرد. حال همین مسئله با استفاده از خواص **Property** ها بسیار ساده‌تر پیاده‌سازی می‌گردد. به مثال زیر توجه نمایید.

مثال 2-10: دسترسی به فیلدهای کلاس به استفاده از **Property** ها

```
using System;
public class PropertyHolder
{
    private int someProperty = 0;
    public int SomeProperty
    {
        get
        {
            return someProperty;
        }
        set
        {
            someProperty = value;
        }
    }
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.SomeProperty = 5;
        Console.WriteLine("Property Value: {0}", propHold.SomeProperty);
        return 0;
    }
}
```

مثال 2-10 چگونگی ایجاد و استفاده از ویژگی‌ها (**Property**) را نشان می‌دهد. کلاس **PropertyHolder** دارای پیاده‌سازی از ویژگی **SomeProperty** است. توجه نمایید که اولید حرف از نام ویژگی با حرف بزرگ نوشته شده و این تنها تفاوت میان اسم ویژگی **SomeProperty** و فیلد **someProperty** می‌باشد. ویژگی دارای دو **accessor** با نامهای **get** و **set** است. **get accessor** مقدار فیلد **someProperty** را باز می‌گرداند. **set accessor** نیز با استفاده از مقدار **value**، مقداری را به **someProperty** تخصیص می‌دهد. کلمه **value** که در **set accessor** آورده شده است جزو کلمات رزرو شده زبان **C#** می‌باشد.

کلاس **PropertyTester** از ویژگی **someProperty** مربوط به کلاس **PropertyHolder** استفاده می‌کند. اولین خط در متد **Main()** شی‌ای از نوع **PropertyHolder** با نام **propHold** ایجاد می‌نماید. سپس مقدار فیلد **someProperty** مربوط به شیء **propHold**، با استفاده از ویژگی **SomeProperty** به 5 تغییر می‌یابد و ملاحظه می‌نمایید که مسئله به همین سادگی است و تنها کافی است تا مقدار مورد نظر را به ویژگی تخصیص دهیم. پس از آن، متد **Console.WriteLine()** مقدار فیلد **someProperty** شیء **propHold** را چاپ می‌نماید. این عمل با استفاده از ویژگی **SomeProperty** شیء **propHold** صورت می‌گیرد. ویژگی‌ها را می‌توان طوری ایجاد نمود که فقط خواندنی (**Read-Only**) باشند. برای این منظور تنها کافیست تا در ویژگی فقط از **get accessor** استفاده نماییم. به مثال زیر توجه نمایید.

ویژگیهای فقط خواندنی (**Read-Only Properties**)

مثال 3-10: ویژگیهای فقط خواندنی

```
using System;
public class PropertyHolder
{
    private int someProperty = 0;
    public PropertyHolder(int propVal)
    {
        someProperty = propVal;
    }
    public int SomeProperty
    {
        get
        {
            return someProperty;
        }
    }
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder(5);
        Console.WriteLine("Property Value: {0}", propHold.SomeProperty);
        return 0;
    }
}
```

مثال 3-10 چگونگی ایجاد یک ویژگی فقط خواندنی را نشان می‌دهد. کلاس **PropertyHolder** دارای ویژگی **SomeProperty** است که فقط **get accessor** را پیاده‌سازی می‌کند. این کلاس **PropertyHolder** دارای سازنده‌ایست که پارامتری از نوع **int** دریافت می‌نماید.

متد **Main()** از کلاس **PropertyTester** شیء جدیدی از **PropertyHolder** با نام **propHold** ایجاد می‌نماید. این نمونه از کلاس **PropertyHolder** از سازنده آن که مقداری صحیح را بعنوان پارامتر دریافت می‌کند، استفاده می‌کند. در این مثال این مقدار برابر با 5 در نظر گرفته می‌شود. این امر باعث تخصیص داده شدن عدد 5 به فیلد **someProperty** از شیء **propHold** می‌شود.

تا زمانیکه ویژگی **SomeProperty** از کلاس **PropertyHolder** فقط خواندنی است، هیچ راهی برای تغییر مقدار فیلد **someProperty** وجود ندارد. بعنوان مثال در صورتیکه عبارت **propHold.SomeProperty = 7** را در کد برنامه اضافه نمایید، برنامه شما کامپایل نخواهد شد چراکه ویژگی **SomeProperty** فقط خواندنی است. اما اگر از این ویژگی در متد **Console.WriteLine()** استفاده نمایید بخوبی کار خواهد کرد زیرا این دستور تنها یک فرآیند خواندن است و با استفاده از **get accessor** این عمل قابل اجرا است.

ویژگیهای فقط نوشتنی (*Write-Only Properties*)

به مثال زیر توجه فرمایید :

مثال 4-10 : ویژگیهای فقط خواندنی

```
using System;
public class PropertyHolder
{
    private int someProperty = 0;
    public int SomeProperty
    {
        set
        {
            someProperty = value;
            Console.WriteLine("someProperty is equal to {0}", someProperty);
        }
    }
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.SomeProperty = 5;
        return 0;
    }
}
```

مثال 4-10 چگونگی ایجاد و استفاده از ویژگی فقط نوشتنی را نشان می‌دهد. در این حالت **get accessor** را از ویژگی **SomeProperty** حذف کرده و به جای آن **set accessor** را قرار داده‌ایم.

متد **Main()** کلاس **PropertyTester** شی ای جدید از همین کلاس با سازنده پیش فرض آن ایجاد می نماید. سپس با استفاده از ویژگی **SomeProperty** از شیء **propHold** مقدار 5 را به فیلد **someProperty** مربوط به شیء **propHold** تخصیص می دهد. در این حالت **set accessor** مربوط به ویژگی **SomeProperty** فراخوانی شده و مقدار 5 را به فیلد **someProperty** تخصیص می دهد و سپس عبارت **"someProperty is equal to 5"** را در خروجی نمایش می دهد.

خلاصه

در این درس با ویژگیها آشنا شدید و نحوه استفاده از آنها را فرا گرفتید. روشهای کلاسیک کپسوله کردن از طریق استفاده از متدهای مجزا صورت می گرفت ولی با استفاده از ویژگیها (**Property**) می توان به اجزای یک شیء همانند یک فیلد دسترسی پیدا کرد. ویژگیها را می توان به صورت فقط خواندنی و یا فقط نوشتنی نیز ایجاد نمود. با استفاده از ویژگیها دسترسی مستقیم به فیلدهای مورد نظر از یک کلاس از بین رفته و این دسترسی تنها از طریق ویژگی مورد نظر امکان پذیر می گردد.

درس یازدهم – اندیکسرها در C# (Indexers)

در این درس با اندیکسرها در C# آشنا می شویم. اهداف این درس به شرح زیر می باشند :

- پیاده سازی اندیکسر
- سرریزی اندیکسرها (Overload)
- درک چگونگی پیاده سازی اندیکسرها با چند پارامتری
- خلاصه
- نکات مهم و مطالب کمکی در زمینه اندیکسرها

اندیکسرها

اندیکسرها مفهومی بسیار ساده در زبان C# هستند. با استفاده از آنها می توانید از کلاس خود همانند یک آرایه استفاده کنید. در داخل کلاس مجموعه ای از مقادیر را به هر طریقی که مورد نظرتان هست مدیریت کنید. این اشیاء می توانند شامل مجموعه ای از اعضای کلاس، یک آرایه دیگر، و یا مجموعه ای از ساختارهای پیچیده داده ای باشند، جدا از پیاده سازی داخلی کلاس، داده های این ساختارها از طریق استفاده از اندیکسرها قابل دسترسی هستند. به مثالی در این زمینه توجه کنید :

مثال 11-1: نمونه ای از یک اندیکسر

```
using System;
/// <summary>
/// مثالی ساده از یک اندیکسر
/// </summary>
class IntIndexer
```



```

{
    private string[] myData;
    public IntIndexer(int size)
    {
        myData = new string[size];
        for (int i=0; i < size; i++)
        {
            myData[i] = "empty";
        }
    }
    public string this[int pos]
    {
        get
        {
            return myData[pos];
        }
        set
        {
            myData[pos] = value;
        }
    }
    static void Main(string[] args)
    {
        int size = 10;
        IntIndexer myInd = new IntIndexer(size);
        myInd[9] = "Some Value";
        myInd[3] = "Another Value";
        myInd[5] = "Any Value";
        Console.WriteLine("\nIndexer Output\n");
        for (int i=0; i < size; i++)
        {
            Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
        }
    }
}

```

مثال 1-11 نحوه پیاده‌سازی اندیکسر را نشان می‌دهد. کلاس **IntIndexer** دارای آرایه رشته‌ای بنام **myData** می‌باشد. این آرایه، عنصری خصوصی (**private**) است و کاربران خارجی (**external users**) نمی‌توانند به آن دسترسی داشته باشند. این آرایه درون سازنده (**constructor**) کلاس تخصیص‌دهی می‌گردد که در آن پارامتر **size** از نوع **int** دریافت می‌شود، از آرایه **myData** نمونه‌ای جدید ایجاد می‌گردد، سپس هر یک از المانهای آن با کلمه "empty" مقداردهی می‌گردد.

عضو بعدی کلاس، اندیکسر است که بوسیله کلمه کلیدی **this** و دو براکت تعریف شده است، **this[int pos]**. این اندیکسر پارامتر موقعیتی **pos** را دریافت می‌نماید. همانطور که حتماً تا کنون دریافته‌اید پیاده‌سازی اندیکسر بسیار شبیه به پیاده‌سازی یک ویژگی (**property**) است. اندیکسر نیز دارای **accessor** های **set** و **get** است که دقیقاً همانند **property** عمل می‌کنند. همانطور که در اعلان این اندیکسر نیز مشاهده می‌شود، متغیری از نوع رشته‌ای را باز می‌گرداند.

در متد **Main()** شیء جدیدی از **IntIndexer** ایجاد شده است و مقادیری به آن افزوده می‌شود و سپس نتایج چاپ می‌گردند. خروجی این برنامه به شکل زیر است :

Indexer Output

```
myInd[0]: empty
myInd[1]: empty
myInd[2]: empty
myInd[3]: Another Value
myInd[4]: empty
myInd[5]: Any Value
myInd[6]: empty
myInd[7]: empty
myInd[8]: empty
myInd[9]: Some Value
```

استفاده از **integer** جهت دسترسی به آرایه‌ها در اغلب زبانهای برنامه‌سازی رایج است ولی زبان **C#** چیزی فراتر از آنرا نیز پشتیبانی می‌کند. در **C#** اندیکسرها را می‌توان با چندین پارامتر تعریف کرد و هر پارامتر می‌تواند از نوع خاصی باشد. پارامترهای مختلف بوسیله کاما از یکدیگر جدا می‌شوند. پارامترهای مجاز برای اندیکسر عبارتند از : **integer**، **enum** و **string**. علاوه بر آن، اندیکسرها قابل سرریزی (**Overload**) هستند. در [مثال 2-11](#) تغییراتی در مثال قبل ایجاد کرده‌ایم تا برنامه قابلیت دریافت اندیکسرها را نیز داشته باشد.

سرریزی اندیکسرها

مثال 2-11 : اندیکسرها سرریز شده (**Overloaded Indexers**)

```
using System;
/// <summary>
/// پیاده‌سازی اندیکسرها سرریز شده
/// </summary>
class OvrIndexer
{
    private string[] myData;
    private int arrSize;

    public OvrIndexer(int size)
    {
        arrSize = size;
```

```

myData = new string[size];
for (int i=0; i < size; i++)
{
    myData[i] = "empty";
} //end of for
} //end of constructor

public string this[int pos]
{
    get
    {
        return myData[pos];
    }
    set
    {
        myData[pos] = value;
    }
} //end of indexer

public string this[string data]
{
    get
    {
        int count = 0;
        for (int i=0; i < arrSize; i++)
        {
            if (myData[i] == data)
            {
                count++;
            } //end of if
        } //end of for
        return count.ToString();
    } //end of get
    set
    {
        for (int i=0; i < arrSize; i++)
        {
            if (myData[i] == data)
            {
                myData[i] = value;
            } //end of if
        } //end of for
    } //end of set
} //end of overloaded indexer

static void Main(string[] args)

```

```

{
    int size = 10;

    OvrIndexer myInd = new OvrIndexer(size);
    myInd[9] = "Some Value";
    myInd[3] = "Another Value";
    myInd[5] = "Any Value";
    myInd["empty"] = "no value";
    Console.WriteLine("\nIndexer Output\n");
    for (int i=0; i < size; i++)
    {
        Console.WriteLine("myInd[{0}]: {1}", i, myInd[i]);
    } //end of for
    Console.WriteLine("\nNumber of \"no value\" entries: {0}",
myInd["no value"]);
} //end of Main()
} //end of class

```

مثال 11-2 نحوه سرریز کردن اندیکسر را نشان می‌دهد. اولین اندیکسر که دارای پارامتری از نوع **int** تحت عنوان **pos** است دقیقاً مشابه **مثال 11-1** است ولی در اینجا اندیکسر جدیدی نیز وجود دارد که پارامتری از نوع **string** دریافت می‌کند. **get accessor** اندیکسر جدید رشته‌ای را برمی‌گرداند که نمایشی از تعداد آیت‌هایی است که با پارامتر مقداری **data** مطابقت می‌کند. **set accessor** مقدار هر یک از مقادیر ورودی آرایه را که مقدارش با پارامتر **data** مطابقت نماید را به مقداری که به اندیکسر تخصیص داده می‌شود، تغییر می‌دهد.

رفتار (**behavior**) اندیکسر سرریز شده که پارامتری از نوع **string** دریافت می‌کند، در متد **Main()** نشان داده شده است. در اینجا **set accessor** مقدار **"No value"** را به تمام اعضای کلاس **myInd** که مقدارشان برابر با **"empty"** بوده است، تخصیص می‌دهد. این **accessor** از دستور زیر استفاده نموده است: **myInd["empty"] = "No value"**. پس از اینکه تمامی اعضای کلاس **myInd** چاپ شدند، تعداد اعضای که حاوی **"No value"** بوده‌اند نیز نمایش داده می‌شوند. این امر با استفاده از دستور زیر در **get accessor** روی می‌دهد: **myInd["No value"]**. خروجی برنامه بشکل زیر است:

```

Indexer Output
myInd[0]: no value
myInd[1]: no value
myInd[2]: no value
myInd[3]: Another Value
myInd[4]: no value
myInd[5]: Any Value
myInd[6]: no value
myInd[7]: no value
myInd[8]: no value

```

myInd[9]: Some Value Number of "no value" entries: 7

علت همزیستی هر دو اندیکسر در [مثال 2-11](#) در یک کلاس مشابه، تفاوت اثرگذاری و فعالیت آنهاست. اثرگذاری و تفاوت اندیکسرها از تعداد و نوع پارامترهای موجود در لیست پارامترهای اندیکسر مشخص می‌گردد. در هنگام استفاده از اندیکسرها نیز، کلاس با استفاده از تعداد و نوع پارامترهای اندیکسرها، می‌تواند تشخیص دهد که در یک فراخوانی از کدام اندیکسر باید استفاده نماید. نمونه‌ای از پیاده‌سازی اندیکسری با چند نوع پارامتر در زیر آورده شده است:

```
public object this[int param1, ..., int paramN]
{
    get
    {
        // process and return some class data
    }
    set
    {
        // process and assign some class data
    }
}
```

خلاصه:

هم اکنون شما با اندیکسرها و نحوه پیاده‌سازی آنها آشنا شده‌اید. با استفاده از اندیکسرها می‌توان به عناصر یک کلاس همانند یک آرایه دسترسی پیدا کرد. در این مبحث اندیکسرها را سرریز شده و چند پارامتری نیز مورد بررسی قرار گرفتند. در آینده و در مباحث پیشرفته‌تر با موارد بیشتری از استفاده اندیکسرها آشنا خواهید شد.

نکات:

1. منظور از اندیکسر سرریز شده چیست؟

هنگامیکه از دو یا چند اندیکسر درون یک کلاس استفاده می‌کنیم، سرریزی (**Overloading**) اندیکسرها رخ می‌دهد. در هنگام فراخوانی اندیکسرها، کلاس تنها از روی نوع بازگشتی اندیکسر و تعداد پارامترهای آن متوجه می‌شود که منظور فراخواننده استفاده از کدام اندیکسر بوده است.

2. از اندیکسر چگونه مانند آرایه استفاده می‌شود؟

همانطور که در این درس مشاهده کردید دسترسی به عناصر اندیکسر همانند آرایه‌ها با استفاده از یک اندیس صورت می‌پذیرد. با استفاده از این اندیس می‌توان به عنصر مورد نظر کلاس دسترسی پیدا نمود.

3. یک مثال عملی استفاده از اندیکسرها چیست؟

یک نمونه بسیار جالب از استفاده اندیکسرها کنترل **ListBox** است. (**ListBox** عنصری است کنترلی که با استفاده از آن لیستی از عناصر رشته‌ای نمایش داده می‌شوند و کاربر با انتخاب یکی از این گزینه‌ها با برنامه ارتباط برقرار می‌کند. در حقیقت این عنصر کنترلی یکی از روشهای دریافت اطلاعات از کاربر است با این تفاوت که در این روش ورودی‌هایی که کاربر می‌تواند وارد نماید محدود شده هستند و از قبل تعیین شده‌اند. نمونه‌ای از یک **ListBox** قسمت انتخاب نوع فونت در برنامه **Word** است که در آن لیستی از فونت‌های موجود در سیستم نمایش داده می‌شود و کاربر با انتخاب یکی از آنها به برنامه اعلام می‌کند که قصد استفاده از کدام فونت سیستم را دارد.) **ListBox** نمایشی از ساختمان داده ایست شبیه به آرایه که اعضای آن همگی از نوع **string** هستند. علاوه بر این این کنترل می‌خواهد تا در هنگام انتخاب یکی از گزینه‌هایش بتواند اطلاعات خود را بطور خودکار **update** نماید و یا به عبارتی بتواند ورودی دریافت نماید. تمامی این اهداف با استفاده از اندیکسر میسر می‌شود. اندیکسرها شبیه به **property** ها اعلان می‌شوند با این تفاوت مهم که اندیکسرها بدون نام هستند و نام آنها تنها کلمه کلیدی **this** است و همین **this** مورد اندیکس شدن قرار می‌گیرد و سایر موارد بشکل پارامتر به اندیکسر داده می‌شوند.

```
public class ListBox: Control
```

```
{  
    private string[] items;  
  
    public string this[int index]  
    {  
        get  
        {  
            return items[index];  
        }  
        set  
        {  
            items[index] = value;  
            Repaint();  
        }  
    }  
}
```

با نگاه به نحوه استفاده از اندیکسر بهتر می‌توان با مفهوم آن آشنا شد. برای مثال دسترسی به **ListBox** بشکل زیر است :

```
ListBox listBox = ...;  
listBox[0] = "hello";  
Console.WriteLine(listBox[0]);
```

نمونه برنامه‌ای که در آن نحوه استفاده از اندیکسر در عنصر کنترلی **ListBox** نشان داده شده، در زیر آورده شده است :

Csharp-Persian_Indexer_Demo

using System;

public class ListBoxTest

{

// رشته‌های مورد نظر به **ListBox** تخصیص داده می‌شوند.

public ListBoxTest(params **string**[] initialStrings)

{

// فضایی را برای ذخیره‌سازی رشته‌های تخصیص می‌دهد.

strings = **new** **String**[256];

// رشته‌های وارد شده به سازنده را درون آرایه‌ای کپی می‌کند.

foreach (**string** s in initialStrings)

{

strings[ctr++] = s;

}

} //end of constructor

// رشته‌ای به انتهای کنترل افزوده می‌شود.

public void Add(**string** theString)

{

if (ctr >= strings.Length)

{

// در این قسمت می‌توان کدی جهت کنترل پر شدن فضای تخصیص داده شده قرار داد.

}

else

strings[ctr++] = theString;

} //end of Add()

// اعلان اندیکسر

public string this[**int** index]

{

get

{

if (index < 0 || index >= strings.Length)

{

// در این قسمت می‌توان کدی جهت کنترل پر شدن فضای تخصیص داده شده قرار داد.

}

return strings[index];

} //end of get

set

{

```

    if (index >= ctr )
    {
        // فراخوانی متدی جهت کنترل خطا
    }
    else
        strings[index] = value;
    } //end of set
} //end of indexer

// تعداد رشته‌های موجود را نشان می‌دهد
public int GetNumEntries( )
{
    return ctr;
}

private string[] strings;
private int ctr = 0;
} //end of ListBoxTest class

public class Tester
{
    static void Main( )
    {
        // ساخت یک ListBox جدید و تخصیص دهی آن
        ListBoxTest lbt = new ListBoxTest("Hello", "World");
        // رشته‌های مورد نظر به کنترل افزوده می‌شوند.
        lbt.Add("Who");
        lbt.Add("Is");
        lbt.Add("John");
        lbt.Add("Galt");
        // رشته جدیدی در خانه شماره یک قرار داده می‌شود.
        string subst = "Universe";
        lbt[1] = subst;
        // کلیه آیتم‌های موجود نمایش داده می‌شوند.
        for (int i = 0; i < lbt.GetNumEntries( ); i++)
        {
            Console.WriteLine("lbt[{0}]: {1}", i, lbt[i]);
        }
    } //end of Main()
} //end of Tester class

```

خروجی نیز بشکل زیر می‌باشد :

Output:

```
lbt[0]: Hello  
lbt[1]: Universe  
lbt[2]: Who  
lbt[3]: Is  
lbt[4]: John  
lbt[5]: Galt
```

توجه :

مطالب انتهایی این درس کمی پیشرفته تر و پیچیده تر از مطالب قبل به نظر می آیند. این انتظار وجود ندارد که شما کلیه مطالب این قسمت را بطور کامل متوجه شده باشید، بلکه هدف تنها آشنا شدن شما با مسایل پیچیده تر و واقعی تر است. در آینده ای نه چندان دور، در سایت به صورت حرفه ای کلیه مطالب و سرفصل های گفته شده را مورد بررسی قرار خواهیم داد. در ابتدا هدف من آشنایی شما با کلیه مفاهیم پایه ای زبان C# است تا بعد از این آشنایی به طور کامل و بسیار پیشرفته به بررسی کلیه مفاهیم زبان پردازیم. پس از اتمام آموزش اولیه تحولات اساسی در سایت مشاهده خواهید کرد و در آن هنگام به بررسی کامل هر مبحث با مثال هایی بسیار واقعی و کاربردی خواهیم پرداخت.

درس دوازدهم – ساختارها در C# (Struct)

در این درس با ساختارها (Struct) در زبان C# آشنا می شویم. اهداف این درس بشرح زیر می باشند

- یک **struct** یا ساختار (Structure) چیست؟
- پیاده سازی ساختارها (Struct)
- استفاده از ساختارها (Struct)
- نکات مهم و مطالب کمکی درباره struct ها

ساختار (struct) چیست؟

همانطور که با استفاده از کلاسها می توان انواع (types) جدید و مورد نظر را ایجاد نمود، با استفاده از **struct** ها می توان انواع مقداری (value types) جدید و مورد نظر را ایجاد نمود. از آنجائیکه **struct** ها بعنوان انواع مقداری در نظر گرفته می شوند، از اینرو تمامی اعمال مورد استفاده بر روی انواع مقداری را می توان برای **struct** ها در نظر گرفت. **struct** ها بسیار شبیه به کلاسها هستند و می توانند دارای فیلد، متد و **property** باشند. عموماً ساختارها مجموعه کوچکی از عناصری هستند که منطقی با یکدیگر دارای رابطه می باشند. برای نمونه می توان به ساختار **Point** موجود در **Framework SDK** اشاره کرد که حاوی دو **property** با نامهای **X** و **Y** است.

با استفاده از ساختارها (**struct**) می‌توان اشیایی با انواع جدید ایجاد کرد که این اشیاء می‌توانند شبیه به انواع موجود (**int, float, ...**) باشند. حال سوال اینست که چه زمانی از ساختارها (**struct**) بجای کلاس استفاده می‌کنیم؟ در ابتدا به نحوه استفاده از انواع موجود در زبان **C#** توجه نمایید. این انواع دارای مقادیر و عملگرهای معینی جهت کار با این مقادیر هستند. حال اگر نیاز به شی‌ای دارید که همانند این انواع رفتار نمایند لازم است تا از ساختارها (**struct**) استفاده نمایید. در ادامه این مبحث نکات و قوانینی را ذکر می‌کنیم که با استفاده از آنها بهتر بتوانید از ساختارها (**struct**) استفاده نمایید.

اعلان و پیاده‌سازی **struct**

برای اعلان یک **struct** کافیست تا با استفاده از کلمه کلیدی **struct** که بدنال آن نام مورد نظر برای ساختار آمده استفاده کرد. بدنه ساختار نیز بین دو کروشه باز و بسته {} قرار خواهد گرفت. به مثال زیر توجه نمایید :

مثال 12-1 : نمونه‌ای از یک ساختار (**Struct**)

```
using System;
struct Point
{
    public int x;
    public int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point Add(Point pt)
    {
        Point newPt;
        newPt.x = x + pt.x;
        newPt.y = y + pt.y;
        return newPt;
    }
}
/// <summary>
/// struct   مثالی از اعلان و ساخت یک
/// </summary>
class StructExample
{
    static void Main(string[] args)
    {
        Point pt1 = new Point(1, 1);
        Point pt2 = new Point(2, 2);
        Point pt3;
        pt3 = pt1.Add(pt2);
        Console.WriteLine("pt3: {0}:{1}", pt3.x, pt3.y);
    }
}
```


}

مثال 12-1 نحوه ایجاد و استفاده از **struct** را نشان می‌دهد. به راحتی می‌توان گفت که یک نوع (**type**)، یک **struct** است، زیرا از کلمه کلیدی **struct** در اعلان خود بهره می‌گیرد. ساختار پایه‌ای یک ساختار پایه‌ای یک **struct** بسیار شبیه به یک کلاس است، ولی تفاوت‌هایی با آن دارد که این تفاوت‌ها در پاراگراف بعدی مورد بررسی قرار می‌گیرند. ساختار **Point** دارای سازنده ایست که مقادیر داده شده با آنرا به فیلدهای **x** و **y** تخصیص می‌دهد. این ساختار همچنین دارای متد **Add()** می‌باشد که ساختار **Point** دیگری را دریافت می‌کند و آنرا به **struct** کنونی می‌افزاید و سپس **struct** جدیدی را باز می‌گرداند.

توجه نمایید که ساختار **Point** جدیدی درون متد **Add()** تعریف شده است. توجه کنید که در اینجا همانند کلاس، نیازی به استفاده از کلمه کلیدی **new** جهت ایجاد یک شیء جدید نمی‌باشد. پس از آنکه نمونه جدیدی از یک ساختار ایجاد شد، سازنده پیش فرض (یا همان سازنده بدون پارامترش) برای آن در نظر گرفته می‌شود. سازنده بدون پارامتر کلیه مقادیر فیلدهای ساختار را به مقادیر پیش فرض تغییر می‌دهد. بعنوان مثال فیلدهای صحیح به صفر و فیلدهای **Boolean** به **false** تغییر می‌کنند. تعریف سازنده بدون پارامتر برای یک ساختار صحیح نمی‌باشد. (یعنی شما نمی‌توانید سازنده بدون پارامتر برای یک **struct** تعریف کنید).

ساختارها (**structs**) با استفاده از عملگر **new** نیز قابل نمونه‌گیری هستند (هر چند نیازی به استفاده از این عملگر نیست). در مثال 12-1 **pt1** و **pt2** که ساختارهایی از نوع **Point** هستند، با استفاده از سازنده موجود درون ساختار **Point** مقداردهی می‌شوند. سومین ساختار از نوع **Point**، **pt3** است و از سازنده بدون پارامتر استفاده می‌کند زیرا در اینجا مقدار آن اهمیتی ندارد. سپس متد **Add()** از ساختار **pt1** فراخوانده می‌شود و ساختار **pt2** را بعنوان پارامتر دریافت می‌کند. نتیجه به **pt3** تخصیص داده می‌شود، این امر نشان می‌دهد که یک ساختار می‌تواند همانند سایر انواع مقداری مورد استفاده قرار گیرد. خروجی مثال 12-1 در زیر نشان داده شده است :

pt3 : 3 : 3

یکی دیگر از تفاوت‌های ساختار و کلاس در اینست که ساختارها نمی‌توانند دارای تخریب کننده (**destructor**) باشند. همچنین ارث‌بری در مورد ساختارها معنی ندارد. البته امکان ارث‌بری بین ساختارها و **interface** ها وجود دارد. یک **interface** نوع مرجعی زبان **C#** است که دارای اعضای بدون پیاده‌سازی است. هر کلاس و یا ساختاری که از یک **interface** ارث‌بری نماید باید تمامی متدهای آنرا پیاده‌سازی کند. درباره **interface** ها در آینده صحبت خواهیم کرد.

خلاصه :

هم اکنون شما با چگونگی ایجاد یک ساختار آشنا شدید. هنگامیکه قصد دارید نوعی را بصورت ساختار یا کلاس پیاده‌سازی کنید، باید به این نکته توجه کنید که این نوع چگونه مورد استفاده قرار می‌گیرد. اگر می‌خواهید سازنده‌ای بدون پارامتر داشته باشید، در اینصورت کلاس تنها گزینه شماست. همچنین توجه نمایید از آنجائیکه یک ساختار بعنوان یک نوع مقداری در نظر گرفته می‌شود، در پشته (**Stack**) ذخیره می‌شود و حال آنکه کلاس در **heap** ذخیره می‌گردد.

همانطور که بطور مختصر در بالا نیز اشاره شد، از نظر نوشتاری **struct (syntax)** و کلاس بسیار شبیه به یکدیگر هستند اما دارای تفاوت‌های بسیار مهمی با یکدیگر می‌باشند.

همانطور که قبلاً نیز اشاره شد شما نمی‌توانید برای یک **struct** سازنده‌ای تعریف کنید که بدون پارامتر است، یعنی برای ایجاد سازنده برای یک **struct** حتماً باید این سازنده دارای پارامتر باشد. به قطعه کد زیر توجه کنید :

```
struct Time
{
    // خطای زمان کامپایل رخ می‌دهد
    public Time() { ... }
    :
}
```

پس از اجرای کد فوق کامپایلر خطایی را ایجاد خواهد کرد بدین عنوان که سازنده **struct** حتماً باید دارای پارامتر باشد. حال اگر بجای **struct** از کلمه کلیدی **class** استفاده کرده بودیم این کد خطایی را ایجاد نمی‌کرد. در حقیقت تفاوت در اینست که در مورد **struct**، کامپایلر اجازه ایجاد سازنده پیش فرض جدیدی را به شما نمی‌دهد ولی در مورد کلاس چنین نیست. هنگام اعلان کلاس در صورتیکه شما سازنده پیش فرضی اعلان نکرده باشید، کامپایلر سازنده‌ای پیش فرض برای آن در نظر می‌گیرد ولی در مورد **struct** تنها سازنده پیش فرضی معتبر است که کامپایلر آنرا ایجاد نماید نه شما !

یکی دیگر از تفاوت‌های بین کلاس و **struct** در آن است که، اگر در کلاس برخی از فیلدهای موجود در سازنده کلاس را مقداردهی نکنید، کامپایلر مقدار پیش فرض صفر، **false** و یا **null** را برای آن فیلد در نظر خواهد گرفت ولی در **struct** تمامی فیلدهای سازنده باید بطور صریح مقداردهی شوند و در صورتیکه شما فیلدی را مقداردهی نکند کامپایلر هیچ مقداری را برای آن در نظر نخواهد گرفت و خطای زمان کامپایل رخ خواهد داد. بعنوان مثال در کد زیر اگر **Time** بصورت کلاس تعریف شده بود خطایی رخ نمی‌داد ولی چون بصورت **struct** تعریف شده خطای زمان کامپایل رخ خواهد داد :

```
struct Time
{
    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
    } // خطای زمان کامپایلی بدین صورت رخ می‌دهد
    :
    private int hours, minutes, seconds;
}
```

تفاوت دیگر کلاس و **struct** در اینست که در کلاس می‌توانید در هنگام اعلان فیلدها را مقداردهی کنید حال آنکه در **struct** چنین عملی باعث ایجاد خطای زمان کامپایل خواهد شد. همانند کدهای فوق، در کد زیر اگر از کلاس بجای **struct** استفاده شده بود خطا رخ نمی‌داد:

```
struct Time
{
    :
    private int hours = 0; // کامپایل رخ می‌دهد خطای زمان
    private int minutes;
    private int seconds;
}
```

آخرین تفاوت بین کلاس و **struct** که ما به آن خواهیم پرداخت در مورد ارث‌بری است. کلاسها می‌توانند از کلاس پایه خود ارث‌بری داشته باشند در حالیکه ارث‌بری در **struct** ها معنایی ندارد و یک **struct** تنها می‌تواند از واسطهها (**interface**) ارث‌بری داشته باشد.

پس از ایجاد یک ساختار چگونه می‌توان از آن استفاده نمود؟

همانطور که گفتیم، ساختارها روشی برای ایجاد انواع جدید مقدار (**Value Types**) هستند. از اینرو پس از ایجاد یک ساختار می‌توان از آن همانند سایر انواع مقداری استفاده نمود. برای استفاده از یک ساختار ایجاد شده کفایت نام آنرا قبل از متغیر مورد نظر قرار دهیم تا متغیر مورد نظر از نوع آن ساختار خاص تعریف شود.

```
struct Time
{
    :
    private int hours, minutes, seconds;
}

class Example
{
    public void Method(Time parameter)
    {
        Time localVariable;
        :
    }
    private Time field;
}
```

آخرین نکته‌ای که در مورد ساختارها برای چندمین بار اشاره می‌کنم اینست که، ساختارها انواع مقداری هستند و مستقیماً مقدار را در خود نگه می‌دارند و از اینرو در **stack** نگه‌داری می‌شوند. استفاده از ساختارها همانند سایر انواع مقداری است.

درس سیزدهم – واسطها (Interfaces)

در این درس با واسطها در زبان C# آشنا خواهیم شد. اهداف این درس بشرح زیر می‌باشند :

- 1- آشنایی با مفهوم کلی واسطها
- 2- تعریف یک واسط
- 3- استفاده از یک interface
- 4- پیاده‌سازی ارث‌بری در interface ها
- 5- نکات مهم و پیشرفته
- 6- مثالی کاربردی از واسطها
- 7- منابع مورد استفاده

واسطها از لحاظ ظاهری بسیار شبیه به کلاس هستند با این تفاوت که دارای هیچ گونه پیاده‌سازی نمی‌باشند. تنها چیزی که در **interface** به چشم می‌خورد تعاریفی نظیر رخداده‌ها، متدها، اندیکسرها و یا **property** ها است. یکی از دلایل اینکه واسطها تنها دارای تعاریف هستند و پیاده‌سازی ندارند آنست که یک **interface** می‌توان توسط چندین کلاس یا **property** مورد ارث‌بری قرار گیرد، از اینرو هر کلاس یا **property** خواستار آنست که خود به پیاده‌سازی اعضا بپردازد.

حال باید دید چرا با توجه به اینکه **interface** ها دارای پیاده‌سازی نیستند مورد استفاده قرار می‌گیرند یا بهتر بگوییم سودمندی استفاده از **interface** ها در چیست؟ تصور کنید که در یک برنامه با مولفه‌هایی سروکار دارید که متغیرند ولی دارای فیلدها یا متدهایی با نامهای یکسانی هستند و باید نام این متدها نیز یکسان باشد. با استفاده از یک **interface** مناسب می‌توان تنها متدها و یا فیلدهای مورد نظر را اعلان نمود و سپس کلاسها و یا **property** های مورد از آن **interface** ارث‌بری نمایند. در این حالت تمامی کلاسها و **property** ها دارای فیلدها و یا متدهایی همان هستند ولی هر یک پیاده‌سازی خاصی از آنها را اعمال می‌نمایند.

نکته مهم دیگر درباره **interface** ها، استفاده و کاربرد آنها در برنامه‌های بزرگی است که برنامه‌ها و یا اشیاء مختلفی در تماس و تراکنش (**transact**) هستند. تصور کنید کلاسی در یک برنامه با کلاسی دیگر در برنامه‌ای دیگر در ارتباط باشد. فرض کنید این کلاس متدی دارد که مقداری از نوع **int** بازمیگرداند. پس از مدتی طراح برنامه به این نتیجه می‌رسد که استفاده از **int** پاسخگوی مشکلش نیست و باید از **long** استفاده نماید. حال شرایط را در نظر بگیرید که برای تغییر یک چنین مسئله ساده‌ای چه مشکل بزرگی پیش خواهد آمد. تمامی فیلدهای مرتبط با این متد باید تغییر داده شوند. در ضمن از مسئله **side effect** نیز نمی‌توان چشم پوشی کرد. (تاثیرات ناخواسته و غیر منتظره و یا به عبارتی پیش بینی نشده که متغیر یا فیلدی بر روی متغیر یا فیلدی دیگر اعمال می‌کند، در اصطلاح **side effect** گفته می‌شود). حال فرض کنید که در ابتدا **interface** ای طراحی شده بود. در صورت اعمال جزئیترین تغییر در برنامه مشکل تبدیل

int به **long** قابل حل بود، چراکه کاربر یا برنامه و در کل **user** برنامه در هنگام استفاده از یک **interface** با پیاده‌سازی پشت پرده آن کاری ندارد و یا بهتر بگوئیم امکان دسترسی به آن را ندارد. از اینرو اعمال تغییرات درون آن تاثیری بر رفتار کاربر نخواهد داشت و حتی کاربر از آن مطلع نیز نمی‌شود. در مفاهیم کلی شیء گرایی، **interface** ها یکی از مهمترین و کاربردی ترین اجزاء هستند که در صورت درک صحیح بسیار مفید واقع می‌شوند. یکی از مثالهای مشهود درباره **interface** ها (البته در سطحی پیشرفته تر و بالاتر) رابطهای کاربر گرافیکی (**GUI**) هستند. کاربر تنها با این رابط سروکار دارد و کاری به نحوه عملیات پشت پرده آن ندارد و اعمال تغییرات در پیاده‌سازی **interface** کاربر را تحت تاثیر قرار نمی‌دهد.

از دیدگاه تکنیکی، واسطها بسط مفهومی هستند که از آن به عنوان انتزاع (**Abstract**) یاد می‌کنیم. در کلاسهای انتزاعی (که با کلمه کلید **abstract** مشخص می‌شوند) سازنده کلاس قدر بود تا فرم کلاس خود را مشخص نماید: نام متدها، نوع بازگشتی آنها و تعداد و نوع پارامتر آنها، اما بدون پیاده‌سازی بدنه متد. یک **interface** همچنین می‌تواند دارای فیلدهایی باشد که تمامی آنها **static** و **final** هستند. یک **interface** تنها یک فرم کلی را بدون پیاده‌سازی به نمایش می‌گذارد.

از این دیدگاه، یک واسط بیان می‌دارد که: "این فرم کلی است که تمامی کلاسهای که این واسط را پیاده‌سازی می‌کنند، باید آنرا داشته باشند." از سوی دیگر کلاسها و اشیاء دیگری که از کلاسی که از یک واسط مشتق شده استفاده می‌کنند، می‌دانند که این کلاس حتماً تمامی متدها و اعضای واسط را پیاده‌سازی می‌کند و می‌توانند به راحتی از آن متدها و اعضا استفاده نمایند. پس به طور کلی می‌توانیم بگوئیم که واسطها بمنظور ایجاد یک پروتکل (**protocol**) بین کلاسها مورد استفاده قرار می‌گیرند. (همچنان که برخی از زبانهای برنامه‌سازی بجای استفاده از کلمه کلیدی **interface** از **protocol** استفاده می‌نمایند).

به دلیل اینکه کلاسها و ساختارهایی که از **interface** ها ارث‌بری می‌کنند موظف به پیاده‌سازی و تعریف آنها هستند، قانون و قاعده‌ای در این باره ایجاد می‌گردد. برای مثال اگر کلاس **A** از واسط **IDisposable** ارث‌بری کند، این ضمانت بوجود می‌آید که کلاس **A** دارای متد **Dispose()** است، که تنها عضو **interface** نیز می‌باشد. هر کدی که می‌خواهد از کلاس **A** استفاده کند، ابتدا چک می‌نماید که آیا کلاس **A** واسط **IDisposable** را پیاده‌سازی نموده یا خیر. اگر پاسخ مثبت باشد آنگاه کد متوجه می‌شود که می‌تواند از متد **A.Dispose()** نیز استفاده نماید. در زیر نحوه اعلان یک واسط نمایش داده شده است.

```
interface IMyInterface
{
    void MethodToImplement();
}
```

در این مثال نحوه اعلان واسطی با نام **IMyInterface** نشان داده شده است. یک قاعده (نه قانون!) برای نامگذاری واسطها آنست که نام واسطها را با "I" آغاز کنیم که اختصار کلمه **interface** است. در **interface** این مثال تنها

یک متد وجود دارد. این متد می‌توان هر متدی با انواع مختلف پارامترها و نوع بازگشتی باشد. توجه نمایید همانطور که گفته شد این متد دارای پیاده‌سازی نیست و تنها اعلان شده است. نکته دیگر که باید به آن توجه کنید آنست که این متد به جای داشتن {} به عنوان بلوک خود، دارای ؛ در انتهای اعلان خود می‌باشد. علت این امر آنست که **interface** تنها نوع بازگشتی و پارامترهای متد را مشخص می‌نماید و کلاس یا شی‌ای که از آن ارث می‌برد باید آنرا پیاده‌سازی نماید. مثال زیر نحوه استفاده از این واسط را نشان می‌دهد.

مثال 13-1 : استفاده از واسطها و ارث‌بری از آنها

```
class InterfaceImplementer : IMyInterface
{
    static void Main()
    {
        InterfaceImplementer iImp = new InterfaceImplementer();
        iImp.MethodToImplement();
    }
    public void MethodToImplement()
    {
        Console.WriteLine("MethodToImplement() called.");
    }
}
```

در این مثال، کلاس **InterfaceImplementer** همانند ارث‌بری از یک کلاس، از واسط **IMyInterface** ارث‌بری کرده است. حال که این کلاس از واسط مورد نظر ارث‌بری کرده است، باید، توجه نمایید باید، تمامی اعضای آنرا پیاده‌سازی کند. در این مثال این عمل با پیاده‌سازی تنها عضو واسط یعنی متد **MethodToImplement()** انجام گرفته است. توجه نمایید که پیاده‌سازی متد باید دقیقا از لحاظ نوع بازگشتی و تعداد و نوع پارامترها شبیه به اعلان موجود در واسط باشد، کوچکترین تغییری باعث ایجاد خطای کامپایلر می‌شود. مثال زیر نحوه ارث‌بری واسطها از یکدیگر نیز نمایش داده شده است.

مثال 13-2 : ارث‌بری واسطها از یکدیگر

```
using System;

interface IParentInterface
{
    void ParentInterfaceMethod();
}

interface IMyInterface : IParentInterface
{
    void MethodToImplement();
}

class InterfaceImplementer : IMyInterface
```

```

{
    static void Main()
    {
        InterfaceImplementer iImp = new InterfaceImplementer();
        iImp.MethodToImplement();
        iImp.ParentInterfaceMethod();
    }

    public void MethodToImplement()
    {
        Console.WriteLine("MethodToImplement() called.");
    }

    public void ParentInterfaceMethod()
    {
        Console.WriteLine("ParentInterfaceMethod() called.");
    }
}

```

مثال 2-13 دارای 2 واسط است: یکی **IMyInterface** و واسطی که از آن ارث می‌برد یعنی **IParentInterface**. هنگامیکه واسطی از واسط دیگری ارث‌بری می‌کند، کلاس یا ساختاری که این واسطها را پیاده‌سازی می‌کند، باید تمامی اعضای واسطهای موجود در سلسله مراتب ارث‌بری را پیاده‌سازی نماید. در مثال 2-13، چون کلاس **InterfaceImplementer** از واسط **IMyInterface** ارث‌بری نموده، پس از واسط **IParentInterface** نیز ارث‌بری دارد، از اینرو باید کلیه اعضای این دو واسط را پیاده‌سازی نماید.

چند نکته مهم:

- 1- با استفاده از کلمه کلید **interface** در حقیقت یک نوع مرجعی (**Reference Type**) جدید ایجاد نموده‌اید.
- 2- از لحاظ نوع ارتباطی که واسطها و کلاسها در ارث‌بری ایجاد می‌نمایند باید به این نکته اشاره کرد که، ارث‌بری از کلاس رابطه "است" یا "بودن" (**is-a relation**) را ایجاد می‌کند (ماشین یک وسیله نقلیه است) ولی ارث‌بری از یک واسط یا **interface** نوع خاصی از رابطه، تحت عنوان "پیاده‌سازی" (**implement relation**) را ایجاد می‌کند. ("می‌توان ماشین را با وام بلند مدت خرید" که در این جمله ماشین می‌تواند خریداری شدن بوسیله وام را پیاده‌سازی کند).
- 3- فرم کلی اعلان **interface** ها به شکل زیر است:

```

[attributes] [access-modifier] interface interface-name
[:base-list]{interface-body}

```

که در اعضای آن بشرح زیر می‌باشند:

attributes: صفتهای واسط

access-modifiers: **private** یا **public** سطح دسترسی به واسط از قبیل

interface-name: نام واسط

لیست واسطه‌هایی که این واسط آنها را بسط می‌دهد. : **base-list**

Interface-body بدنه واسط که در آن اعضای آن مشخص می‌شوند :

توجه نمایید که نمی‌توان یک واسط را بصورت **virtual** اعلان نمود.

4- هدف از ایجاد یک **interface** تعیین توانائی‌هاییست که می‌خواهیم در یک کلاس وجود داشته باشند.

5- به مثالی در زمینه استفاده از واسطها توجه کنید :

فرض کنید می‌خواهید واسطی ایجاد نمایید که متدها و **property** های لازم برای کلاسی را که می‌خواهد قابلیت خواندن و نوشتن از/به یک پایگاه داده یا هر فایلی را داشته باشد، توصیف نماید. برای این منظور می‌توانید از واسط **IStorable** استفاده نمایید.

در این واسط دو متد **Read()** و **Write()** وجود دارند که در بدنه واسط تعریف می‌شوند ک

```
interface IStorable
```

```
{  
    void Read( );  
    void Write(object);  
}
```

حال می‌خواهید کلاسی با عنوان **Document** ایجاد نمایید که این کلاس باید قابلیت خواندن و نوشتن از/به پایگاه داده را داشته باشد، پس می‌توانید کلاس را از روی واسط **IStorable** پیاده‌سازی کنید.

```
public class Document : IStorable  
{  
    public void Read( ) {...}  
    public void Write(object obj) {...}  
    // ...  
}
```

حال بعنوان طراح برنامه، شما وظیفه داری تا به پیاده‌سازی این واسط بپردازید، بطوریکه کلیه نیازهای شما را برآورده نماید. نمونه‌ای از این پیاده‌سازی در مثال 3-13 آورده شده است.

مثال 3-13: پیاده‌سازی واسط و ارث‌بری – مثال کاربردی

```
using System;
```

```
// interface اعلان
```

```
interface IStorable  
{  
    void Read( );  
    void Write(object obj);  
    int Status { get; set; }  
}
```

```
public class Document : IStorable  
{  
    public Document(string s)  
    {  
        Console.WriteLine("Creating document with: {0}", s);  
    }  
}
```

```

    }

    public void Read( )
    {
        Console.WriteLine("Implementing the Read Method for
IStorable");
    }

    public void Write(object o)
    {
        Console.WriteLine("Implementing the Write Method for
IStorable");
    }

    public int Status
    {
        get
        {
            return status;
        }
        set
        {
            status = value;
        }
    }
    private int status = 0;
}

public class Tester
{
    static void Main( )
    {
        Document doc = new Document("Test Document");
        doc.Status = -1;
        doc.Read( );
        Console.WriteLine("Document Status: {0}", doc.Status);
        IStorable isDoc = (IStorable) doc;
        isDoc.Status = 0;
        isDoc.Read( );
        Console.WriteLine("IStorable Status: {0}", isDoc.Status);
    }
}

```

خروجی برنامه نیز بشکل زیر است :

Output:

```

Creating document with: Test Document
Implementing the Read Method for IStorable
Document Status: -1
Implementing the Read Method for IStorable

```

- 6- در مثال فوق توجه نمایید که برای متدها واسط **IStorable** هیچ سطح دسترسی (**public, private** و ...) در نظر گرفته نشده است. در حقیقت تعیین سطح دسترسی باعث ایجاد خطا می شود چراکه هدف اصلی از ایجاد یک واسط ایجاد شیء است که تمامی اعضای آن برای تمامی کلاسها قابل دسترسی باشند.
- 7- توجه نمایید که از روی یک واسط نمی توان نمونه ای جدید ایجاد کرد بلکه باید کلاسی از آن ارث بری نماید.
- 8- کلاسی که از واسط ارث بری می کند باید تمامی متدهای آنرا دقیقا همان گونه که در واسط مشخص شده پیاده سازی نماید. به بیان کلی، کلاسی که از یک واسط ارث می برد، فرم و ساختار کلی خود را از واسط می گیرد و نحوه رفتار و پیاده سازی آنرا خود انجام می دهد.

خلاصه :

در این درس با مفاهیم کلی و اصلی درباره واسطها آشنا شدید. هم اکنون می دانید که واسطها چه هستند و سودمندی استفاده از آنها چیست. همچنین نحوه پیاده سازی واسط و ارث بری از آنرا آموختید. مبحث واسطها بسیار گسترده و مهم است و امید است در بخشهای آینده در سایت، بتوانم تمامی مطالب را بطور حرفه ای و کامل در اختیار شما قرار دهم.

درس چهاردهم – رخدادها و *delegate* ها در C#

نکته مهم قبل از مطالعه این درس

توجه نمایید، **delegate** ها و رخدادها بسیار با یکدیگر در تعامل اند، از اینرو در برخی موارد، قبل از آموزش و بررسی رخدادها، به ناچار، از آنها نیز استفاده شده و یا به آنها رجوع شده است. رخدادها در قسمت انتهایی این درس مورد بررسی قرار می گیرند، از اینرو در صورتیکه در برخی موارد دچار مشکل شدید و یا درک مطلب برایتان دشوار بود، ابتدا کل درس را تا انتها مطالعه نمایید و سپس در بار دوم با دیدی جدید به مطالب و مفاهیم موجود در آن نگاه کنید. در اغلب کتابهای آموزشی زبان C# نیز ایندو مفهوم با یکدیگر آورده شده اند ولی درک رخدادها مستلزم درک و فراگیری کامل **delegate** هاست، از اینرو مطالب مربوط به **delegate** ها را در ابتدا قرار داده ام.

هدف ما در این درس به شرح زیر است :

- [مقدمه](#)
- [درک اینکه یک delegate چیست؟](#)
- [اعلان و پیاده سازی delegate ها](#)
- [درک سودمندی delegate ها](#)
- [حل مسئله بدون استفاده از delegate](#)
- [حل مسئله با استفاده از delegate](#)
- [اعلان delegate ها \(بخش پیشرفته\)](#)

- فراخوانی delegate ها (بخش پیشرفته)
- ایجاد نمونه‌های جدید از یک delegate (بخش پیشرفته)
- درک اینکه یک رخداد یا یک event چیست؟
- اعلان رخدادها
- نکات و توضیحات پیشرفته
- ثبت شدن در یک رخداد
- لغو عضویت در یک رخداد
- فراخوانی رخدادها
- مثالی پیشرفته از استفاده رخدادها در فرمهای ویندوز
- نکات کلیدی درباره رخدادها و delegate ها
- منابع مورد استفاده

طی درسهای گذشته، چگونگی ایجاد و پیاده‌سازی انواع مرجعی (Reference Type) را با استفاده از ساختارهای زبان C#، یعنی کلاسها (Class) و واسطها (Interface)، فرا گرفتید. همچنین فرا گرفتید که با استفاده از این انواع مرجعی، میتوانید نمونه‌های جدیدی از اشیاء را ایجاد کرده و نیازهای توسعه نرم‌افزار خود را تامین نمایید. همانطور که تا کنون دیدید، با استفاده از کلاسها قادر به ساخت اشیائی هستید که دارای صفات (Attribute) و رفتارهای (Behavior) خاصی بودند. با استفاده از واسطها، یکسری از صفات و رفتارها را تعریف می‌کردیم تا فرم کلی داشته باشیم و تمام اشیاء خود به پیاده‌سازی این صفا و رفتارها می‌پرداختند. در این درس با یکی دیگر از انواع مرجعی (Reference Type) در زبان C# آشنا خواهید شد.

مقدمه‌ای بر رخدادها و delegate ها

در گذشته، پس از اجرای یک برنامه، برنامه مراحل اجرای خود را مرحله به مرحله اجرا می‌نمود تا به پایان برسد. در صورتیکه نیاز به ارتباط و تراکنش با کاربر نیز وجود داشت، این امر محدود و بسیار کنترل شده صورت می‌گرفت و معمولاً ارتباط کاربر با برنامه تنها پر کردن و یا وارد کردن اطلاعات خاصی در فیلدهایی مشخص بود.

امروزه با پیشرفت کامپیوتر و گسترش تکنولوژیهای برنامه نویسی و با ظهور رابطهای کاربر گرافیکی (GUI) ارتباط بین کاربر و برنامه بسیار گسترش یافته و دیگر این ارتباط محدود به پر کردن یکسری فیلد نیست، بلکه انواع عملیات از سوی کاربر قابل انجام است. انتخاب گزینه‌ای خاص در یک منو، کلیک کردن بر روی دکمه‌ها برای انجام عملیاتی خاص و ...، رهیافتی که امروزه در برنامه‌نویسی مورد استفاده است، تحت عنوان "برنامه‌نویسی بر پایه رخدادها" (Event-Based Programming) شناخته می‌شود. در این رهیافت برنامه همواره منتظر انجام عملی از سوی کاربر می‌ماند و پس از انجام عملی خاص، رخداد مربوط به آن را اجرا می‌نماید. هر عمل کاربر باعث اجرای رخدادی می‌شود. در این میان

برخی از رخدادها بدون انجام عملی خاص از سوی کاربر اجرا می‌شوند، همانند رخدادهای مربوط به ساعت سیستم که مرتباً در حال اجرا هستند.

رخدادها (Events) بیان این مفهوم هستند که در صورت اتفاق افتادن عملی در برنامه، کاری باید صورت گیرد. در زبان C# مفاهیم Event و Delegate دو مفهوم بسیار وابسته به یکدیگر هستند و با یکدیگر در تعامل می‌باشند. برای مثال، مواجهه با رخدادها و انجام عمل مورد نظر در هنگام اتفاق افتادن یک رخداد، نیاز به یک event handler دارد تا در زمان بروز رخداد، بتوان به آن مراجعه نمود. Event handler ها در C# معمولاً با delegate ها ساخته می‌شوند. از delegate ، می‌توان به عنوان یک Callback یاد نمود، بدین معنا که یک کلاس می‌تواند به کلاسی دیگر بگوید: "این عمل خاص را انجام بده و هنگامیکه عملیات را انجام دادی منرا نیز مطلع کن". با استفاده از delegate ها، همچنین می‌توان متدهایی تعریف نمود که تنها در زمان اجرا قابل دسترسی باشند.

Delegate

Delegate ها، یکی دیگر از انواع مرجعی زبان C# هستند که با استفاده از آنها می‌توانید مرجعی به یک متد داشته باشید، بدین معنا که delegate ها، آدرس متدی خاص را در خود نگه میدارند. در صورتیکه قبلاً با زبان C برنامه‌نویسی کرده‌اید، حتماً با این مفهوم آشنایی دارید. در زبان C این مفهوم با اشاره‌گرها (pointer) بیان می‌شود. اما برای افرادی که با زبانهای دیگری برنامه‌نویسی می‌کرده‌اند و با این مفهوم مانوس نیستند، شاید این سوال مطرح شود که چه نیازی به داشتن آدرس یک متد وجود دارد. برای پاسخ به این سوال اندکی باید تامل نمایید.

بطور کلی می‌توان گفت که delegate نوعی است شبیه به متد و همانند آن نیز رفتار می‌کند. در حقیقت delegate انتزاعی (Abstraction) از یک متد است. در برنامه‌نویسی ممکن به شرایطی برخورد کرده باشید که در آنها می‌خواهید عمل خاصی را انجام دهید اما دقیقاً نمی‌دانید که باید چه متد یا شیء‌ای را برای انجام آن عمل خاص مورد استفاده قرار دهید. در برنامه‌های تحت ویندوز این گونه مسائل مشهودتر هستند. برای مثال تصور کنید در برنامه شما دکمه‌ای قرار دارد که پس از فشار دادن این دکمه توسط کاربر شیء‌ای یا متدی باید فراخوانی شود تا عمل مورد نظر شما بر روی آن انجام گیرد. می‌توان بجای اتصال این دکمه به شیء یا متد خاص، آنرا به یک delegate مرتبط نمود و سپس آن delegate را به متد یا شیء خاصی در هنگام اجرای برنامه متصل نمود.

ابتدا، به نحوه استفاده از متدها توجه نمایید. معمولاً، برای حل مسایل خود الگوریتم‌هایی طراحی می‌نمایم که این الگوریتم‌های کارهای خاصی را با استفاده از متدها انجام می‌دهد، ابتدا متغیرهایی مقدار دهی شده و سپس متدی جهت پردازش آنها فراخوانی می‌گردد. حال در نظر بگیرید که به الگوریتمی نیاز دارید که بسیار قابل انعطاف و قابل استفاده مجدد (reusable) باشد و همچنین در شرایط مختلف قابلیت‌های مورد نظر را در اختیار شما قرار دهد. تصور کنید، به الگوریتمی نیاز دارید که از نوعی از ساختمان داده پشتیبانی کند و همچنین می‌خواهید این ساختمان داده را در مواردی مرتب (sort) نمایید، بعلاوه می‌خواهید تا این ساختمان داده از انواع مختلفی تشکیل شده باشد. اگر انواع موجود در این ساختمان داده را ندانید، چگونه می‌خواهید الگوریتمی جهت مقایسه عناصر آن طراحی کنید؟ شاید از یک حلقه if/then/else یا دستور switch برای این منظور استفاده کنید، اما استفاده از چنین الگوریتمی محدودیتی برای ما

ایجاد خواهد کرد. روش دیگر، استفاده از یک واسط است که دارای متدی عمومی باشد تا الگوریتم شما بتواند آنرا فراخوانی نماید، این روش نیز مناسب است، اما چون مبحث ما در این درس **delegate** ها هستند، می‌خواهیم مسئله را از دیدگاه **delegate** ها مورد بررسی قرار دهیم. روش حل مسئله با استفاده از آنها اندکی متفاوت است.

روش دیگر حل مسئله آنست که، می‌توان **delegate** ی را به الگوریتم مورد نظر ارسال نمود و اجازه داد تا متد موجود در آن، عمل مورد نظر ما را انجام دهد. چنین عملی در مثال 1-14 نشان داده شده است.

(به صورت مسئله توجه نمایید : می‌خواهیم مجموعه‌ای از اشیاء را که در یک ساختمان داده قرار گرفته‌اند را مرتب نماییم. برای اینکار نیاز به مقایسه این اشیاء با یکدیگر داریم. از آنجائیکه این اشیاء از انواع (**type**) مختلف هستند به الگوریتمی نیاز داریم تا بتواند مقایسه بین اشیاء نظیر را انجام دهد. با استفاده از روشهای معمول این کار امکان پذیر نیست، چراکه نمی‌توان اشیائی از انواع مختلف را با یکدیگر مقایسه کرد. برای مثال شما نمی‌توانید نوع عددی **int** را با نوع رشته‌ای **string** مقایسه نمایید. به همین دلیل با استفاده از **delegate** ها به حل مسئله پرداخته‌ایم. به مثال زیر به دقت توجه نمایید تا بتوانید به درستی مفهوم **delegate** را درک کنید.)

مثال 1-14 : اعلان و پیاده‌سازی یک **delegate**

```
using System;
```

```
// delegate در اینجا اعلان می‌گردد.
```

```
public delegate int Comparer(object obj1, object obj2);
public class Name
{
    public string FirstName = null;
    public string LastName = null;
    public Name(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }
    // delegate method handler
    public static int CompareFirstNames(object name1, object
name2)
    {
        string n1 = ((Name)name1).FirstName;
        string n2 = ((Name)name2).FirstName;
        if (String.Compare(n1, n2) > 0)
        {
            return 1;
        }
        else if (String.Compare(n1, n2) < 0)
        {
            return -1;
        }
        else
        {

```

```

        return 0;
    }
}
public override string ToString()
{
    return FirstName + " " + LastName;
}
}

class SimpleDelegate
{
    Name[] names = new Name[5];
    public SimpleDelegate()
    {
        names[0] = new Name("Meysam", "Ghazvini");
        names[1] = new Name("C#", "Persian");
        names[2] = new Name("Csharp", "Persian");
        names[3] = new Name("Xname", "Xfamily");
        names[4] = new Name("Yname", "Yfamily");
    }
    static void Main(string[] args)
    {
        SimpleDelegate sd = new SimpleDelegate();
        // delegate ساخت نمونه‌ای جدید از
        Comparer cmp = new Comparer(Name.CompareFirstNames);
        Console.WriteLine("\nBefore Sort: \n");
        sd.PrintNames();

        sd.Sort(cmp);
        Console.WriteLine("\nAfter Sort: \n");
        sd.PrintNames();
    }

    public void Sort(Comparer compare)
    {
        object temp;
        for (int i=0; i < names.Length; i++)
        {
            for (int j=i; j < names.Length; j++)
            {
                // از compare همانند یک متد استفاده می‌شود
                if ( compare(names[i], names[j]) > 0 )
                {
                    temp = names[i];
                    names[i] = names[j];
                    names[j] = (Name)temp;
                }
            }
        }
    }
}

```

```

    }
}
public void PrintNames()
{
    Console.WriteLine("Names: \n");
    foreach (Name name in names)
    {
        Console.WriteLine(name.ToString());
    }
}
}

```

اولین اعلان در این برنامه، اعلان **delegate** است. اعلان **delegate** بسیا ر شبیه به اعلان متد است، با این تفاوت که دارای کلمه کلیدی **delegate** در اعلان است و در انتهای اعلان آن ";" قرار می‌گیرد و نیز پیاده‌سازی ندارد. در زیر اعلان **delegate** که در مثال 14-1 آورده شده را مشاهده می‌نمایید:

```
public delegate int Comparer(object obj1, object obj2);
```

این اعلان، مدل متدی را که **delegate** می‌تواند به آن اشاره کند را تعریف می‌نماید. متدی که می‌توان از آن بعنوان **delegate handler** برای **Comparer** استفاده نمود، هر متدی می‌تواند باشد اما حتماً باید پارامتر اول و دوم آن از نوع **object** بوده و مقداری از نوع **int** بازگرداند. در زیر متدی که بعنوان **delegate handler** در مثال 14-1 مورد استفاده قرار گرفته است، نشان داده شده است:

```

public static int ComparerFirstNames(object name1, object
name2)
{
    ...
}

```

برای استفاده از **delegate** می‌بایست نمونه‌ای از آن ایجاد کنید. ایجاد نمونه جدید از **delegate** همانند ایجاد نمونه‌ای جدید از یک کلاس است که به همراه پارامتری جهت تعیین متد **delegate handler** ایجاد می‌شود:

```
Comparer cmp = new Comparer(Name.ComparerFirstName);
```

در مثال 14-1، **cmp** بعنوان پارامتری برای متد **Sort()** مورد استفاده قرار گرفته است. به روش ارسال **delegate** به متد **Sort()** توجه نمایید:

```
sd.Sort(cmp);
```

با استفاده از این تکنیک، هر متد **delegate handler** به سادگی در زمان اجرا به متد **Sort()** قابل ارسال است. برای مثال می‌توان **handler** دیگری با نام **CompareLastNames()** تعریف کنید، نمونه جدیدی از **Comparer** را با این پارامتر ایجاد کرده و سپس آنرا به متد **Sort()** ارسال نمایید.

درک سودمندی *delegate* ها

برای درک بهتر **delegate** ها به بررسی یک مثال می‌پردازیم. در اینجا این مثال را یکبار بدون استفاده از **delegate** و بار دیگر با استفاده از آن حل کرده و بررسی می‌نمائیم. مطالب گفته شده در بالا نیز به نحوی مرور خواهند شد. توجه نمایید، همانطور که گفته شد **delegate** ها و رخدادها بسیار با یکدیگر در تعامل‌اند، از اینرو در برخی موارد به ناچار از رخدادها نیز استفاده شده است. رخدادها در قسمت انتهایی این درس آورده شده‌اند، از اینرو در صورتیکه در برخی موارد دچار مشکل شدید و یا درک مطلب برایتان دشوار بود، ابتدا کل درس را تا انتها مطالعه نمایید و سپس در بار دوم با دیدی جدید به مطالب و مفاهیم موجود در آن نگاه کنید. در اغلب کتابهای آموزشی زبان **C#** نیز ایندو مفهوم با یکدیگر آورده شده‌اند ولی درک رخدادها مستلزم درک و فراگیری کامل **delegate** هاست، از اینرو مطالب مربوط به **delegate** ها را در ابتدا قرار داده‌ام.

ل مسئله بدون استفاده از *delegate*

فرض کنید، میخواهید برنامه بنویسید که عمل خاصی را هر یک ثانیه یکبار انجام دهد. یک روش برای انجام چنین عملی آنست که، کار مورد نظر را در یک متد پیاده‌سازی نمایید و سپس با استفاده از کلاسی دیگر، این متد را هر یک ثانیه یکبار فراخوانی نمائیم. به مثال زیر توجه کنید :

```
class Ticker
{
    :
    public void Attach(Subscriber newSubscriber)
    {
        subscribers.Add(newSubscriber);
    }
    public void Detach(Subscriber exSubscriber)
    {
        subscribers.Remove(exSubscriber);
    }
    // هر ثانیه فراخوانی میگردد
    private void Notify()
    {
        foreach (Subscriber s in subscribers)
        {
            s.Tick();
        }
    }
    :
    private ArrayList subscribers = new ArrayList();
}
class Subscriber
{
    public void Tick()
    {
        :
    }
}
```

```

    }
}
class ExampleUse
{
    static void Main()
    {
        Ticker pulsed = new Ticker();
        Subscriber worker = new Subscriber();
        pulsed.Attach(worker);
        :
    }
}

```

این مثال مطمئناً کار خواهد کرد اما ایدآل و بهینه نیست. اولین مشکل آنست که کلاس **Ticker** بشدت وابسته به **Subscriber** است. به بیان دیگر تنها نمونه‌های جدید کلاس **Subscriber** می‌توانند از کلاس **Ticker** استفاده نمایند. اگر در برنامه کلاس دیگری داشته باشید که بخواهید آن کلاس نیز هر یک ثانیه یکبار اجرا شود، می‌بایست کلاس جدیدی شبیه به **Ticker** ایجاد کنید. برای بهینه کردن این مسئله می‌توانید از یک واسطه (**Interface**) نیز کمک بگیرید. برای این منظور می‌توان متد **Tick** را درون واسطی قرار داد و سپس کلاس **Ticker** را به این واسط مرتبط نمود.

```

interface Tickable
{
    void Tick();
}

class Ticker
{
    public void Attach(Tickable newSubscriber)
    {
        subscribers.Add(newSubscriber);
    }
    public void Detach(Tickable exSubscriber)
    {
        subscribers.Remove(exSubscriber);
    }
    // هر ثانیه فراخوانی میگردد
    private void Notify()
    {
        foreach (Tickable t in subscribers)
        {
            t.Tick();
        }
    }
    :
    private ArrayList subscribers = new ArrayList();
}

```

```
}
```

این راه حل این امکان را برای کلیه کلاسها فراهم می‌نماید تا واسط **Tickable** را پیاده‌سازی کنند.

```
class Clock : Tickable
{
    :
    public void Tick()
    {
        :
    }
    :
}
class ExampleUse
{
    static void Main()
    {
        Ticker pulsed = new Ticker();
        Clock wall = new Clock();
        pulsed.Attach(wall);
        :
    }
}
```

حال به بررسی همین مثال با استفاده از **delegate** خواهیم پرداخت.

حل مسئله با استفاده از *delegate*

استفاده از واسطها در برنامه‌ها، مطمئناً روشی بسیار خوب است، اما کامل نبوده اشکالاتی دارد. مشکل اول آنست که این روش بسیار کلی و عمومی است. تصور نمایید می‌خواهید از تعداد زیادی از سرویسها استفاده نمایید (عنوان مثال در برنامه‌های مبتنی بر **GUI**. در اینگونه برنامه‌ها حجم عظیمی از رخدادهای وجود دارند که می‌بایست با تمامی آنها در ارتباط باشید.) مشکل دیگر آنست که استفاده از واسط، بدین معناست که متد **Tick** باید متدی **public** باشد، از اینرو هر کدی می‌تواند **Clock.Tick** را در هر زمانی فراخوانی نماید. روش مناسب تر آنست که مطمئن شویم تنها اعضای خاص قادر به فراخوانی و دسترسی به **Clock.Tick** هستند. با استفاده از **delegate** تمامی این امکانات برای ما فراهم خواهد شد و برنامه‌هایی با ایمنی بالاتر و پایدارتر می‌توانیم داشته باشیم.

اعلان *Delegate*

در مثال ما، متد **Tick** از واسط **Tickable** از نوع **void** بود و هیچ پارامتری دریافت نمی‌کرد :

```
interface Tickable
{
    void Tick();
}
```

برای این متد می‌توان **delegate** ی تعریف نمود که ویژگیهای آنرا داشته باشد :

```
delegate void Tick();
```

همانطور که قبلاً نیز گفته شد، این عمل نوع جدیدی را ایجاد می‌نماید که می‌توان از آن همانند سایر انواع استفاده نمود. مثلاً می‌توان آنرا بعنوان پارامتری برای یک متد در نظر گرفت :

```
void Example(Tick param)
{
    :
}
```

فراخوانی *delegate*

قدرت و توانایی **delegate** زمانی مشهود می‌گردد که می‌خواهید از آن استفاده نمایید. برای مثال، با متغیر **param** در مثال قبل چکار می‌توانید انجام دهید؟ اگر **param** متغیری از نوع **int** بود، از مقدار آن استفاده می‌کردید و با استفاده از عملگرهایی نظیر +، - و یا عملگرهای مقایسه‌ای، عملی خاص را بر روی آن انجام می‌دادید. اما حال که **param** متغیری از نوع **int** نیست، چه می‌کنید؟ متغیر **param** یک **delegate** است و همانطور که گفته شد، **delegate** انتزاعی از یک متد است، پس هر عملی که متد انجام می‌دهد، **delegate** نیز می‌تواند انجام دهد. با استفاده از پرانتز، می‌توان از **delegate** استفاده نمود :

```
void Example(Tick param)
{
    param();
}
```

نکته : همانطور که اشاره شد، **delegate** یکی از انواع مرجعی است از اینرو مقدار آن می‌تواند برابر با **Null** باشد. در مثال فوق، اگر مقدار **param** برابر با **Null** باشد، کامپایلر خطای **NullReferenceException** را ایجاد می‌نماید.

همانند متدها، **delegate** ها باید بطور کامل و صحیح فراخوانی گردند. با توجه به اعلان **Tick**، در زمان فراخوانی این **delegate**، مثلاً **param**، باید توجه داشت که هیچ پارامتری را نمی‌توان به آن ارسال نمود و نمی‌توان آنرا به متغیری نسبت داد چراکه این **delegate** بصورت **void** اعلان شده و مقدار بازگشتی ندارد.

```
void Example(Tick param)
{
    param(42); // خطای زمان کامپایل رخ می‌دهد
    int hhg = param(); // خطای زمان کامپایل رخ می‌دهد
    Console.WriteLine(param()); // خطای زمان کامپایل رخ می‌دهد
}
```

توجه نمایید که **delegate** را به هر نحوی می‌توانید اعلان نمایید. برای مثال به نسخه دیگری از **Tick** توجه کنید :

```
delegate void Tick(int hours, int minutes, int seconds);
```

اما به یاد داشته باشید که همانند متد، در هنگام استفاده از آن باید پارامترهای صحیح به آن ارسال نمایید :

```
void Example(Tick method)
{
    method(12, 29, 59);
}
```

با استفاده از delegate می‌توانید کلاس Ticker را پیاده‌سازی کنید :

```
delegate void Tick(int hours, int minutes, int seconds);
class Ticker
{
    :
    public void Attach(Tick newSubscriber)
    {
        subscribers.Add(newSubscriber);
    }
    public void Detach(Tick exSubscriber)
    {
        subscribers.Remove(exSubscriber);
    }

    private void Notify(int hours, int minutes, int
seconds)
    {
        foreach (Tick method in subscribers)
        {
            method(hours, minutes, seconds);
        }
    }
    :
    private ArrayList subscribers = new ArrayList();
}
```

ساخت نمونه‌های جدید از یک *delegate*

آخرین کاری که باید انجام دهید، ایجاد نمونه‌های جدید از **delegate** ساخته شده است. یک نمونه جدید از یک **delegate**، تنها انتزاعی از یک متد است که با نامگذاری آن متد ایجاد می‌شود.

```
class Clock
{
    :
    public void RefreshTime(int hours, int minutes, int sec
onds)
    {
        :
    }
    :
}
```

با توجه به ساختار Tick، ملاحظه می‌نمایید که متد RefreshTime کاملاً با این **delegate** همخوانی دارد :

```
delegate void Tick(int hours, int minutes, int seconds);
```


و این بدین معناست که می‌توان نمونه جدید از **Tick** ایجاد کرد که انتزاعی از فراخوانی **RefreshTime** در شیء خاصی از **Clock** است.

```
Clock wall = new Clock();
```

```
:
```

```
Tick m = new Tick(wall.RefreshTime);
```

حال که **m**، ایجاد شد، می‌توانید از آن بصورت زیر استفاده نمایید :

```
m(12, 29, 59);
```

این دستور در حقیقت کار دستور زیر را انجام می‌دهد (چون **m** دقیقاً انتزاع آن است) :

```
wall.RefreshTime(12, 29, 59);
```

همچنین می‌توانید **m** را بعنوان پارامتر به متدی ارسال نمایید. حال تمام چیزهایی را که برای حل مسئله با استفاده از **delegate** بدانها نیاز داشتیم را بررسی کردیم. در زیر مثالی را مشاهده می‌کنید که کلاسهای **Tick** و **Clock** را به یکدیگر مرتبط نموده است. در این مثال از واسط استفاده نشده و متد **RefreshTime** متدی **private** است :

```
delegate void Tick(int hours, int minutes, int seconds);
```

```
:
```

```
class Clock
```

```
{
```

```
:
```

```
public void Start()
```

```
{
```

```
    ticking.Attach(new Tick(this.RefreshTime));
```

```
}
```

```
public void Stop()
```

```
{
```

```
    ticking.Detach(new Tick(this.RefreshTime));
```

```
}
```

```
    private void RefreshTime(int hours, int minutes, int seconds)
```

```
{
```

```
        Console.WriteLine("{0}:{1}:{2}", hours, minutes, seconds);
```

```
}
```

```
    private Ticker ticking = new Ticker();
```

```
}
```

با اندکی تامل و صرف وقت می‌توانید **delegate** را بطور کامل درک نمایید.

رخدادها (*Events*)

در برنامه‌های **Console**، برنامه منتظر ورود اطلاعات یا دستوراتی از سوی کاربر می‌ماند و با استفاده از این اطلاعات کار مورد نظر را انجام می‌دهند. این روش برقراری ارتباط با کاربر، روشی ناپایدار و غیر قابل انعطاف است. در مقابل

برنامه‌های **Console**، برنامه‌های مدرن وجود دارند که با استفاده از **GUI** با کاربر در ارتباطند و بر پایه رخدادها بنا شده‌اند (**Event-Based**)، بدین معنا که رخدادی (منظور از رخداد اتفاقی است که در سیستم یا محیط برنامه صورت می‌گیرد) در سیستم روی می‌دهد و بر اساس این رخداد عملی در سیستم انجام می‌شود. در برنامه‌های تحت ویندوز، نیازی به استفاده از حلقه‌های متعدد جهت منتظر ماندن برای ورودی از کاربر نیست، بلکه با استفاده از رخدادها، تراکنش بین سیستم و کاربر کنترل می‌شود.

یک **event** در زبان **C#**، عضوی از کلاس است، که در صورت بروز رخداد خاصی، فعال می‌شود و عملی را انجام می‌دهد. معمولاً برای فعال شده **event** از دو عبارت **fires** و **raised** استفاده می‌شود. هر متدی که بخواهد، میتواند در لیست رخداد ثبت شده و به محض اتفاق افتادن آن رخداد، از آن مطلع گردد.

بطور کلی می‌توان گفت که یک رخداد همانند یک فیلد اعلان می‌شود با این تفاوت مهم که نوع آنها حتماً باید یک **delegate** باشد.

Delegate و رخدادها در کنار یکدیگر کار می‌کنند تا قابلیت‌های یک برنامه را افزایش دهند. این پروسه با شروع یک کلاس که یک رخداد را تعریف می‌کند، آغاز می‌شود. هر کلاسی، که این رخداد را درون خود داشته باشد، در آن رخداد ثبت شده است و می‌تواند متدی را به آن رخداد تخصیص دهد. این عمل با استفاده از **delegate** ها صورت می‌پذیرد، بدین معنی که **delegate** متدی را که برای رخداد ثبت می‌شود را تعیین می‌نماید. **Delegate** ها می‌توانند هر یک از **delegate** های از پیش تعریف شده **Net**. و یا هر **delegate** ی باشند که توسط کاربر تعریف شده است. بطور کلی، **delegate** ی را به رخدادی تخصیص می‌دهیم تا متدی را که بهنگام روی دادن رخداد فراخوانی می‌شود، معین گردد. مثال زیر روش تعریف رخداد را نشان می‌دهد.

مثال 2-14: اعلان و پیاده‌سازی رخدادها

```
using System;

public delegate void MyDelegate();

class Listing14-2
{
    public static event MyDelegate MyEvent;

    static void Main()
    {
        MyEvent += new MyDelegate(CallbackMethod);

        // فراخوانی رخداد
        MyEvent();

        Console.ReadLine();
    }
}
```

```

    }

    public static void CallbackMethod()
    {
        Console.WriteLine("CallbackMethod.");
    }
}

```

در این مثال، ابتدا اعلان یک **delegate** دیده می‌شود. درون کلاس، رخدادی با نام **MyEvent** و از نوع **MyDelegate** تعریف شده است. در متد **Main()** نیز مرجع جدیدی به رخداد **MyEvent** افزوده شده است. همانطور که در این مثال نیز مشاهده می‌کنید، **delegate** ها تنها با استفاده از += می‌توانند به رخدادها افزوده شوند. در این مثال هر گاه **MyEvent** فراخوانی شود، متد **CallbackMethod** اجرا می‌شود چراکه با استفاده از مرجع **delegate** به رخداد مرتبط شده است. (یا در اصطلاح در رخداد ثبت شده است).

مثال فوق را بدون استفاده از رخداد نیز می‌توان نوشت. این نسخه از مثال 2-14 که تنها در آن از **delegate** استفاده شده در زیر آورده شده است :

```

using System;

public delegate void MyDelegate();

class UsingDelegates
{
    static void Main()
    {
        MyDelegate del = new MyDelegate(CallbackMethod);

        // delegate فراخوانی
        del();

        Console.ReadLine();
    }

    public static void CallbackMethod()
    {
        Console.WriteLine("CallbackMethod.");
    }
}

```

باید توجه کنید که موارد کاربرد رخدادها بیشتر در برنامه‌های تحت ویندوز نمایان می‌شود و در اینجا شاید وجود آنها در برنامه برای شما مشهود نباشد. در آینده، به بررسی برنامه‌نویسی فرمهای ویندوز نیز خواهیم رسید و در آنجا به طور مفصل درباره **event** ها و **delegate** ها مجدداً بحث خواهیم نمود.

بطور خلاصه می‌توان گفت، با استفاده از **delegate** ها روشی برای ایجاد دسترسی به متدها بطور پویا را فراهم نمودیم. با استفاده از رخدادها نیز، در صورت بروز اتفاقی خاص، عملی خاص انجام می‌گیرد. این عمل معمولاً با استفاده از یک **delegate** که مرجعی به یک متد در خود دارد انجام می‌گیرد.

توضیحات پیشرفته :

در انتهای این درس می‌خواهم توضیحات پیشرفته تری را نیز در اختیار شما قرار دهم. در قسمت مربوط به **delegate** ها در همین درس، مثالی مطرح شد که در آن **delegate** ی با نام **Tick** وجود داشت. اعلان این **delegate** به صورت زیر بود :

```
delegate void Tick(int hours, int minutes, int seconds);
```

حال می‌خواهیم به این مثال یک رخداد نیز اضافه کنیم. در زیر رخداد **tick** از نوع **Tick** اعلان شده است :

```
class Ticker
{
    public event Tick tick;
    :
}
```

باید توجه نمایید که یک رخداد بطور خودکار لیست اعضای خود را مدیریت می‌کند و نیازی به استفاده از یک مجموعه، مانند آرایه، برای مدیریت اعضای مرتبط با آن نیست. نکته : یک رخداد بطور خودکار خود را تخصیص دهی می‌کند و نیازی به ساخت نمونه جدید از روی یک رخداد وجود ندارد.

عضو شدن در یک رخداد (ثبت شدن در یک رخداد)

برای افزودن **delegate** جدید به یک رخداد کفایت تا از عملگر += استفاده نماییم. مثال زیر کلاس **Clock** را نشان می‌دهد که در آن فیلدی از نوع **Ticker** با نام **pulsed** وجود دارد. کلاس **Ticker** دارای رخداد **tick** از نوع **delegate** ی بنام **Tick** است. متد **Clock.Start**، **delegate** ی از نوع **Tick** را با استفاده از عملگر += به **pulsed.tick** می‌افزاید.

```
delegate void Tick(int hours, int minutes, int seconds);
```

```
class Ticker
{
    public event Tick tick;
    :
}
```

```
class Clock
{
    :
}
```

```

public void Start()
{
    pulsed.tick += new Tick(this.RefreshTime);
}
:
private void RefreshTime(int hours, int minutes, int
seconds)
{
    :
}

```

```

private Ticker pulsed = new Ticker();
}

```

هنگامیکه رخداد **pulsed.tick** اجرا می‌شود، تمامی **delegate** های مرتبط با آن نیز فراخوانی می‌شوند که در اینجا یکی از آنها **RefreshTime** است. (به مثال موجود در بخش **delegate** رجوع نمایید).

خارج شدن از لیست یک رخداد

همانطور که با استفاده از عملگر += می‌توان **delegate** ی را به یک رخداد افزود، با استفاده از عملگر -= نیز می‌توان **delegate** خاصی را از لیست اعضای یک رخداد خارج نمود.

```

class Clock
{
    :
    public void Stop()
    {
        pulsed.tick -= new Tick(this.RefreshTime);
    }
    private void RefreshTime(int hours, int minutes, int se
conds)
    {
        :
    }
    private Ticker pulsed = new Ticker();
}

```

نکته: همانطور که می‌دانید، عملگرهای += و -= بر پایه دو عملگر اصلی + و - ایجاد شده‌اند. از اینرو در مورد **delegate** ها نیز می‌توان از عملگر + استفاده نمود. استفاده از عملگر + برای **delegate** ها باعث ایجاد **delegate** جدیدی می‌شود که به هنگام فراخوانی هر دو **delegate** را به هم فرا می‌خواند.

فراخوانی یک رخداد

یک رخداد نیز همانند **delegate** با استفاده از دو پراپرتی فراخوانی می‌گردد. پس از اینکه رخدادی فراخوانی شد، کلیه **delegate** های مرتبط با آن بترتیب فراخوانی می‌شوند. برای مثال در اینجا کلاس **Ticker** را در نظر بگیرید که دارای متد **private** با نام **Notify** است که رخداد **tick** را فرا می‌خواند:

```

class Ticker

```



```

{
    public event Tick tick;
    :
    private void Notify(int hours, int minutes, int
seconds)
    {
        if (tick != null)
        {
            tick(hours, minutes, seconds);
        }
    }
    :
}

```

نکته مهم: توجه کنید که در مثال فوق چک کردن **null** نبودن رخداد **tick** ضروری است، چراکه فیلد رخداد بطور ضمنی **null** در نظر گرفته می‌شود و تنها زمانی مقداری به غیر **null** می‌گیرد که **delegate** ی به آن مرتبط شده باشد. در صورت فراخوانی رخداد **null** خطای **NullReferenceException** روی خواهد داد.

رخدادها دارای سطح امنیتی داخلی بسیار بالایی هستند. رخدادی که بصورت **public** اعلان می‌شود، تنها از طریق متدها یا عناصر داخل همان کلاس قابل دسترسی است. بعنوان مثال، **tick** رخدادی درون کلاس **Ticker** است، از اینرو تنها متدهای درون **Ticker** می‌توانند **tick** را فرا بخوانند.

```

class Example
{
    static void Main()
    {
        Ticker pulsed = new Ticker();
        pulsed.tick(12, 29, 59); // خطای زمان کامپایل رخ می‌دهد
    }
}

```

مثالی پیشرفته از استفاده رخدادها در فرمهای ویندوز

حال که تا حدودی با رخدادها و ساختار آنها آشنا شدید، در این قسمت قصد داریم تا مقداری درباره استفاده رخدادها در فرمهای ویندوز و **GUI** ها صحبت نمایم. هر چند تا کنون کلیه برنامه‌ها و مطالبی که مشاهده کرده‌اید مبتنی بر **Console** بوده‌اند، اما به علت استفاده بیشمار رخدادها در فرمهای ویندوز و برنامه‌های مبتنی بر **GUI**، لازم دیدم تا مطالبی نیز در این باره بیان کنم. هر چند فرمهای ویندوز و **GUI** مطالبی هستند که خود نیاز به بحث و بررسی دقیق دارند و انشا... در رؤس آتی سایت مورد بررسی قرار خواهند گرفت. در صورتیکه مطالب این قسمت برای شما دشوار و یا گنگ بود نگران و یا ناراحت نشوید چرا که فعلاً برای یادگیری این مطالب آنهم بدون مقدمه اندکی زود است، بیشتر هدف من از این بخش آشنا شدن شما با کاربردهای پیشرفته‌تر رخدادها در برنامه‌نویسی بوده است.

کلاسهای GUI مربوط به **Net Framework**. بطور گسترده‌ای از رخدادها استفاده می‌نمایند. در مثالی که در اینجا مورد بررسی قرار می‌دهیم، برنامه‌ای وجود دارد که دارای یک فرم به همراه دو دکمه (**Button**) بر روی آن است. این دو دکمه بوسیله دو فیلد از نوع **Button** ایجاد می‌شوند. (**Button** عضو **System.Windows.Forms** است). کلاس **Button** از کلاس **Control** ارث‌بری می‌کند و دارای رخدادی با نام **Click** از نوع **EventHandler** است. به مثال توجه نمایید.

```
namespace System
{
    public delegate void EventHandler(object sender,
    EventArgs args);

    public class EventArgs
    {
        :
    }
}
namespace System.Windows.Forms
{
    public class Control :
    {
        public event EventHandler Click;
        :
    }
    public class Button : Control
    {
        :
    }
}
```

توجه نمایید که کد فوق، کد مربوط به **namespace** مربوط به **System** است که نحوه پیاده‌سازی آنرا نشان می‌دهد. همانطور که ملاحظه می‌نمایید، درون **System**، **delegate** ی با نام **EventHandler** تعریف شده است. در زیر این **namespace**، اعلان **System.Windows.Forms** نیز آورده شده تا نحوه اعلان رخداد **Click** و ارث‌بری کلاس **Button** از کلاس **Control** نیز مشخص شود.

پس از اینکه بر روی دکمه‌ای واقع در فرم ویندوز کلیک کنید، **Button** بطور خودکار رخداد **Click** را فرا می‌خواند. هدایت این پروسه باعث می‌شود تا بتوان به سادگی **delegate** ی برای کنترل این رخداد ایجاد نمود. در مثالی که در زیر مشاهده می‌کنید، دکمه‌ای با نام **Okay**، متدی بنام **Okay_Click** و رخدادی جهت اتصال **Okay** به متد **Okay_Click** وجود دارد.

```
class Example : System.Windows.Forms.Form
{
```

```

private System.Windows.Forms.Button okay;
:
public Example()
{
    this.okay = new System.Windows.Forms.Button();
    this.okay.Click += new System.EventHandler(this.oka
y_Click);
    :
}

private void okay_Click(object sender, System.EventsArg
s args)
{
    :
}
}

```

همانطور که مشاهده می‌کنید، کلاس **Example** از **System.Windows.Forms.Form** مشتق می‌شود، از اینرو تمامی خواص آن را به ارث می‌برد. **Okay** نیز از نوع **Button** اعلان شده است. درون سازنده (Constructor) کلاس **Example**، متد **Okay.Click** به رخداد افزوده شده و مرجع **this.Okay.Click** نیز، متد مورد نظر را تعیین نموده است. همانطور که گفته شد، **EventHandler** نیز **delegate** مورد نظر در این مثال است. درون متد **Okay_Click** نیز میتوان کد خاصی را قرار داد تا عمل مورد نظر را انجام دهد. پس از کلیک کردن بر روی دکمه **Okay**، عمل مورد نظری که درون متد **Okay_Click** قرار داده شده، اجرا می‌شود.

این کد، شبیه به کدهایی است که توسط محیط‌های برنامه‌سازی نظیر **Visual Studio.Net** و یا **C#Builder** بطور خودکار تولید می‌شوند و تنها کافیسست تا شما کد مربوط به **Okay_Click** را درون آن وارد نمایید. رخدادهایی که توسط کلاسهای **GUI** تولید می‌شوند همواره از یک الگوی خاص پیروی می‌کنند. این رخدادهای همواره از نوع **delegate** هستند که مقدار بازگشتی ندارد (**void**) و دارای دو آرگومان است. آرگومان اول همیشه فرستنده رخداد و آرگومان دوم همیشه آرگومان **EventArgs** یا کلاس مشتق شده از **EventArgs** است. آرگومان **sender** به شما این امکان را می‌دهد تا از یک **delegate** برای چندین رخداد استفاده نمایید. (بعنوان مثال برای چندین دکمه).

نکاتی چند درباره **delegate** ها و **event** ها

- **Delegate** ها بطور ضمنی از **System.Delegate** ارث‌بری می‌کنند. **Delegate** حاوی متدها، **property** ها و عملگرهایی است که می‌توان آنها را بعنوان پارامتر به متدهای دیگر ارسال نمود. همچنین به دلیل اینکه **System.Delegate** بخشی از **.Net Framework** است، از اینرو **delegate** های ایجاد شده در **C#** را می‌توان در زبانهای دیگری نظیر **Visual Basic.Net** نیز استفاده نمود.

- هنگام اعلان پارامترها برای **delegate**، حتماً باید برای آنها نام در نظر بگیرید و فقط به مشخص کردن نوع پارامترها بسنده نکنید.
- رخدادها عناصر بسیار مفید و پر استفاده‌ای هستند که با بکارگیری **delegate** ها بسیار قدرتمند ظاهر می‌شوند. بدست آوردن مهارت در ایجاد و استفاده از آنها نیاز به تمرین و تفکر بسیار دارد

درس پانزدهم - برخورد با استثناها

(Exception Handling)

در این درس با چگونگی برخورد با استثناها (یا خطاهای غیر قابل پیش‌بینی) در زبان برنامه‌سازی **C#** آشنا می‌شویم. اهداف ما در این درس بشرح زیر می‌باشد :

- (1) درک و فهم صحیح یک استثناء یا **Exception**
- (2) پیاده‌سازی یک روتین برای برخورد با استثناها بوسیله بلوک **try/catch**
- (3) آزادسازی منابع تخصیص داده شده به یک برنامه در یک بلوک **finally**

استثناها، در حقیقت خطاهای غیر منتظره در برنامه‌های ما هستند. اکثراً، می‌توان و باید روشهایی را جهت برخورد با خطاهای موجود در برنامه در نظر گرفت و آنها را پیاده‌سازی کرد. بعنوان مثال، بررسی و تایید داده‌های ورودی کاربران، بررسی اشیاء تهی یا **Null** و یا بررسی نوع بازگشتی متدها، می‌توانند از جمله مواردی باشند که باید مورد بررسی قرار گیرند. این خطاها، خطاهایی معمول و رایجی هستند که اکثر برنامه‌نویسان از آنها مطلع بوده و راههایی را برای بررسی آنها در نظر می‌گیرند تا از وقوع آنها جلوگیری نمایند.

اما زمانهایی وجود دارند که از اتفاق افتادن یک خطا در برنامه بی اطلاع هستید و انتظار وقوع خطا در برنامه را ندارید. بعنوان مثال، هرگز نمی‌توان وقوع یک خطای **I/O** را پیش‌بینی نمود و یا کمبود حافظه برای اجرای برنامه و از کار افتادن برنامه به این دلیل. این موارد بسیار غیر منتظره و ناخواسته هستند، اما در صورت وقوع بهتر است بتوان راهی برای مقابله و برخورد با آنها پیدا کرده و با آنها برخورد نمود. در این جاست که مسئله برخورد با استثناها (**Exception Handling**) مطرح می‌شود.

هنگامیکه استثنایی رخ می‌دهد، در اصطلاح می‌گوئیم که این استثناء، **thrown** شده است. در حقیقت **thrown**، شی‌ای است مشتق شده از کلاس **System.Exception** که اطلاعاتی در مورد خطا یا استثناء رخ داده را نشان می‌دهد. در قسمتهای مختلف این درس با روش مقابله با استثناها با استفاده از بلوک های **try/catch** آشنا خواهید شد.

کلاس **System.Exception** حاوی تعداد بسیار زیادی متد و **property** است که اطلاعات مهمی در مورد استثناء و خطای رخ داده را در اختیار ما قرار می‌دهد. برای مثال، **Message** یکی از **property** های موجود در این کلاس است که اطلاعاتی درباره نوع استثناء رخ داده در اختیار ما قرار می‌دهد. **StackTrace** نیز، اطلاعاتی در مورد **Stack** (پشته) و محل وقوع خطا در **Stack** در اختیار ما قرار خواهد داد.

تشخیص چنین استثناهایی، دقیقاً با روتین‌های نوشته شده توسط برنامه‌نویس در ارتباط هستند و بستگی کامل به الگوریتمی دارد که وی برای چنین شرایطی در نظر گرفته است. برای مثال، در صورتیکه با استفاده از متد `System.IO.File.OpenRead()` اقدام به باز کردن فایلی نماییم، احتمال وقوع **(Thrown)** یکی از استثنای زیر وجود دارد :

SecurityException
ArgumentException
ArgumentNullException
PathTooLongException
DirectoryNotFoundException
UnauthorizedAccessException
FileNotFoundException
NotSupportedException

با نگاهی بر مستندات **Net Framework SDK**، به سادگی می‌توان از خطاها و استثناهایی که ممکن است یک متد ایجاد کند، مطلع شد. تنها کافیسست به قسمت **Reference/Class Library** رفته و مستندات مربوط به **Namespace/Class/Method** را مطالعه نمایید. در این مستندات هر خطا دارای لینکی به کلاس تعریف کننده خود است که با استفاده از آن می‌توان متوجه شد که این استثناء به چه موضوعی مربوط است. پس از اینکه از امکان وقوع خطایی در قسمتی از برنامه مطلع شدید، لازم است تا با استفاده از مکانیزمی صحیح به مقابله با آن پردازید. هنگامیکه یک استثناء در اصطلاح **thrown** می‌شود (یا اتفاق می‌افتد) باید بتوان به طریقی با آن مقابله نمود. با استفاده از بلوکهای **try/catch** می‌توان چنین عملی را انجام داد. پیاده‌سازی این بلوکها بدین شکل هستند که، کدی را که احتمال تولید استثناء در آن وجود دارد را در بلوک **try**، و کد مربوط به مقابله با این استثناء رخ داده را در بلوک **catch** قرار می‌دهیم. در مثال [15-1](#) چگونگی پیاده‌سازی یک بلوک **try/catch** نشان داده شده است. بدلیل اینکه متد **OpenRead()** احتمال ایجاد یکی از استثنای گفته شده در بالا را دارد، آنرا در بلوک **try** قرار داده ایم. در صورتیکه این خطا رخ دهد، با آن در بلوک **catch** مقابله خواهیم کرد. در مثال [15-1](#) در صورت بروز استثناء، پیغامی در مورد استثناء رخ داده و اطلاعاتی در مورد محل وقوع آن در **Stack** برای کاربر بر روی کنسول نمایش داده می‌شود. نکته : توجه نمایید که کلیه مثالهای موجود در این درس به طور تعمدی دارای خطاهایی هستند تا شما با نحوه مقابله با استنهاها آشنا شوید.

```
using System;
using System.IO;

class TryCatchDemo
{
    static void Main(string[] args)
    {
        try
```



```

    {
        File.OpenRead("NonExistentFile");
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

```

هر چند کد موجود در مثال [15-1](#) تنها داری یک بلوک **catch** است، اما تمامی استثنائاتی که ممکن است رخ دهند را نشان داده و مورد بررسی قرار می‌دهد زیرا از نوع کلاس پایه استثناء، یعنی **Exception** تعریف شده است. در کنترل و مقابله با استثناءها، باید استثنائات خاص را زودتر از استثنائات کلی مورد بررسی قرار داد. کد زیر نحوه استفاده از چند بلوک **catch** را نشان می‌دهد:

```

catch(FileNotFoundException fnfex)
{
    Console.WriteLine(fnfex.ToString());
}

```

```

catch(Exception ex)
{
    Console.WriteLine(ex.ToString());
}

```

در این کد، در صورتیکه فایل مورد نظر وجود نداشته باشد، **FileNotFoundException** رخ داده و توسط اولین بلوک **catch** مورد بررسی قرار می‌گیرد. اما در صورتیکه **PathTooLongException** رخ دهد، توسط دومین بلوک **catch** بررسی خواهد شد. علت آنست که برای **PathTooLongException** بلوک **catch** ای در نظر گرفته نشده است و تنها گزینه موجود جهت بررسی این استثناء بلوک کلی **Exception** است. نکته ای که در اینجا باید بدان توجه نمود آنست که هرچه بلوکهای **catch** مورد استفاده خاص تر و جزئی تر باشند، پیغامها و اطلاعات مفیدتری در مورد خطا می‌توان بدست آورد.

استثنائاتی که مورد بررسی قرار نگیرند، در بالای **Stack** نگهداری می‌شوند تا زمانی که بلوک **try/catch** مناسبی مربوط به آنها یافت شود. در صورتیکه برای استثناء رخ داده بلوک **try/catch** در نظر گرفته نشده باشد، برنامه متوقف شده و پیغام خطایی ظاهر می‌گردد. این چنین حالتی بسیار نامناسب بوده و کاربران را دچار آشفتگی خواهد کرد. استفاده از روشهای مقابله با استثناءها در برنامه، روشی مناسب و رایج است و باعث قدرتمند تر شدن برنامه می‌شود. یکی از حالتها بسیار خطرناک و نامناسب در زمان وقوع استثناءها، هنگامی است که استثناء یا خطای رخ داده باعث از کار افتادن برنامه شود ولی منابع تخصیص داده شده به آن برنامه آزاد نشده باشند. هر چند بلوک **catch** برای برخورد با

استثناها مناسب است ولی در مورد گفته شده نمی تواند کمکی به حل مشکل نماید. برای چنین شرایطی که نیاز به آزادسازی منابع تخصیص داده شده به یک برنامه داریم، از بلوک **finally** استفاده می کنیم.

کد نشان داده شده در مثال [2-15](#)، به خوبی روش استفاده از بلوک **finally** را نشان می دهد. همانطور که حتماً می دانید، رشته های فایلی پس از اینکه کار با آنها به اتمام می رسد باید بسته شوند، در غیر اینصورت هیچ برنامه دیگری قادر به استفاده از آنها نخواهد بود. در این حالت، رشته فایلی، منبعی است که می خواهیم پس از باز شدن و اتمام کار، بسته شده و به سیستم باز گردد. در مثال [2-15](#)، **outStream** با موفقیت باز می شود، بدین معنا که برنامه **handle** ای به یک فایل باز شده در اختیار دارد. اما زمانیکه می خواهیم **inStream** را باز کنیم، استثناء **FileNotFoundException** رخ داده و باعث می شود که کنترل برنامه سریعاً به بلوک **catch** منتقل گردد.

در بلوک **catch** می توانیم فایل **outStream** را ببندیم. اما برنامه تنها زمانی به بلوک **catch** وارد می شود که استثنایی رخ دهد. پس اگر هیچ استثنایی رخ نداده و برنامه به درستی عمل نماید، فایل باز شده **outStream** هرگز بسته نشده و یکی از منابع سیستم به آن بازگردانده نمی شود. بنابراین باید برای بستن این فایل نیز فکری کرد. این کاری است که در بلوک **finally** رخ می دهد. بدین معنا که در هر حالت، چه برنامه با استثنایی روبرو شود و چه نشود، قبل از خروج از برنامه فایل باز شده، بسته خواهد شد. در حقیقت می توان گفت بلوک **finally**، بلوکی است که تضمین می نماید در هر شرایطی اجرا خواهد شد. پس برای حصول اطمینان از اینکه منابع مورد استفاده برنامه پس از خروج برنامه، به سیستم باز گردانده می شوند، می توان از این بلوک استفاده کرد.

```
using System;
using System.IO;

class FinallyDemo
{
    static void Main(string[] args)
    {
        FileStream outputStream = null;
        FileStream inputStream = null;
        try
        {
            outputStream = File.OpenWrite("DestinationFile.txt");
            inputStream = File.OpenRead("BogusInputFile.txt");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
        finally
        {
            if (outputStream != null)
            {
                outputStream.Close();
                Console.WriteLine("outStream closed.");
            }
        }
    }
}
```

```

    }
    if (inStream != null)
    {
        inStream.Close();
        Console.WriteLine("inStream closed.");
    }
}
}
}

```

استفاده از بلوک **finally** الزامی نیست، اما روشی مناسب برای بالا بردن کارایی برنامه است. ممکن است سوالی در اینجا مطرح شود: در صورتیکه پس از بلوک **catch** و بدون استفاده از بلوک **finally**، فایل باز شده را ببندیم، باز هم منبع تخصیص داده شده به برنامه آزاد می شود. پس چه دلیلی برای استفاده از بلوک **finally** وجود دارد؟ در پاسخ به این سوال باید گفت، در شرایط نرمال که تمامی برنامه بطور طبیعی اجرا می شود و اتفاق خاصی رخ نمی دهد، می توان گفت که دستورات بعد از بلوک **catch** اجرا شده و منبع تخصیص داده شده به سیستم آزاد می شود. اما برای بررسی همیشه باید بدترین حالت را در نظر گرفت. فرض کنید درون خود بلوک **catch** استثنائی رخ دهد که شما آنرا پیش بینی نکرده اید و یا این استثناء باعث متوقف شدن برنامه شود، در چنین حالتی کدهای موجود بعد از بلوک **catch** هرگز اجرا نخواهند شد و فایل همچنان باز می ماند. اما با استفاده از بلوک **finally** می توان مطمئن بود که کد موجود در این بلوک حتماً اجرا شده و منبع تخصیص داده شده به برنامه آزاد می گردد.

در اینجا به پایان درس پانزدهم رسیدیم. هم اکنون می بایست درک صحیحی از استثناء بدست آورده باشید. همچنین می توانید به سادگی الگوریتمهایی جهت بررسی استثناءها بوسیله بلوکهای **try/catch** پیاده سازی نمایید. بعلاوه می توانید با استفاده از بلوک **finally** مطمئن باشید که منابع تخصیص داده شده به برنامه، به سیستم باز خواهند گشت چراکه این بلوک حتماً اجرا می شود و می توان کدهای مهمی را که می خواهیم تحت هر شرایطی اجرا شوند را درون آن قرار داد

درس شانزدهم – استفاده از صفتها در C#

در این درس با نحوه استفاده از صفتها در زبان C# آشنا خواهید شد. اهداف ما در این درس به شرح زیر است:

- 1- صفتها چه هستند و چرا از آنها استفاده می کنیم
- 2- استفاده از صفتهای تک پارامتری و چند پارامتری
- 3- انواع پارامترهای صفت (پارامترهای Named و Positional)
- 4- Target های صفتها (عناصری که صفتها بر روی آنها اعمال می شوند)
- 5- تولید صفتهای شخصی
- 6- تعریف و یا کنترل موارد استفاده از یک صفت
- 7- استفاده از پارامترهای Named و Positional در صفتهای شخصی
- 8- انواع (type) معتبر برای پارامترهای صفت

9- استفاده از صفتها در زمان اجرا

10- خلاصه مطالب

11- منابع

صفتها در حقیقت اطلاعات توضیحی هستند که می‌توانید آنها را به برنامه‌های خود بیفزایید. صفتها را می‌توان برای کلیه عناصر برنامه از قبیل کلاسها، واسطها، اسمبلی‌ها و ... مورد استفاده قرار داد. از این اطلاعات می‌توان برای موارد متنوعی در زمان اجرای برنامه استفاده نمود. برای مثال می‌توان به صفتی مانند `DllImportAttribute` اشاره کرد که امکان برقراری ارتباط با توابع کتابخانه‌ای `Win32` را فراهم می‌نماید. همچنین صفتهایی نیز وجود دارند که برنامه‌نویس یا توسعه‌دهنده برنامه را در امر تولید برنامه یاری می‌نمایند. برای مثال می‌توان به صفت `ObsoleteAttribute` اشاره کرد که با استفاده از آن، در زمان کامپایل برنامه پیغامی برای برنامه‌نویس نمایش داده می‌شود و مشخص می‌کند که متدی خاص مورد استفاده قرار نگرفته و یا دیگر مورد استفاده نیست. همچنین هنگامیکه با فرمهای ویندوز کار می‌کنیم، صفتهای بسیاری وجود دارند که امکان استفاده از این فرمها را فراهم کرده و باعث می‌شوند تا اطلاعات مربوط به این عناصر در `property` فرم ظاهر شوند. یکی دیگر از موارد استفاده از صفتها در مسایل امنیتی اسمبلی‌های `.Net` است. برای مثال صفتهایی وجود دارند که باعث جلوگیری از فراخوانی‌های غیر مجاز می‌شوند، بدین معنی که تنها اجازه فراخوانی را به متدها یا اشیایی می‌دهند که قبلاً تعریف شده و مشخص شده باشند.

یکی از علت‌های استفاده از صفتها آنست که، اغلب سرویس‌هایی را که آنها برای کاربر فراهم می‌نمایند، بسیار پیچیده است و با کدهای معمولی نمی‌توان آنها را بدست آورد. از اینرو استفاده از صفتها در بسیاری از موارد ضروری و اجتناب ناپذیر است. همانطور که خواهید دید، صفتها به برنامه‌های ما `Metadata` اضافه می‌نمایند. پس از کامپایل برنامه‌های `C#`، فایل اسمبلی برای آن ایجاد می‌گردد که این اسمبلی معمولاً یا یک فایل اجرایی است و یا یک `Dll` است. توصیف اسمبلی، در `Metadata` ی مربوط به آن قرار می‌گیرد. طی پروسه‌ای تحت عنوان `Reflection`، صفت یک برنامه از طریق فایل `Metadata` موجود در اسمبلی آن قابل دسترسی می‌گردد. (برای آشنایی بیشتر با اسمبلی و `Metadata` می‌توانید به "کامپایل یک برنامه سی شارپ" در همین سایت مراجعه نمایید.) در حقیقت صفتها، کلاس‌هایی هستند که می‌توانید آنها را با زبان `C#` تولید کرده و جهت افزودن اطلاعاتی توضیحی به کد خود، از آنها استفاده نمایید. این اطلاعات در زمان اجرای برنامه از طریق `Reflection` قابل دسترسی هستند.

در این درس با روش استفاده از صفتها و چگونگی ارتباط دادن آنها با عناصر مختلف برنامه آشنا خواهید شد.

مفاهیم اولیه درباره صفتها

صفتها را معمولاً قبل از اعلان عنصر مورد نظر در برنامه قرار می‌دهند. اعلان صفتها بدین صورت است که نام صفت درون دو براکت قرار می‌گیرد.

[ObsoleteAttribute]

استفاده از کلمه `Attribute` در اعلان صفت الزامی نیست، از اینرو اعلان زیر با اعلان فوق یکسان است :

[Obsolete]

همچنین صفتها می‌توانند دارای پارامتر نیز باشند که با استفاده از آنها خواص بیشتری را در اختیار برنامه قرار می‌دهند. در [مثال 16-1](#) موارد متنوعی از استفاده صفت *ObsoleteAttribute* را مشاهده می‌نمایید.

مثال 16-1: نحوه استفاده از صفتها

```
using System;

class BasicAttributeDemo
{
    [Obsolete]
    public void MyFirstDeprecatedMethod()
    {
        Console.WriteLine("Called
MyFirstDeprecatedMethod().");
    }

    [ObsoleteAttribute]
    public void MySecondDeprecatedMethod()
    {
        Console.WriteLine("Called
MySecondDeprecatedMethod().");
    }

    [Obsolete("You shouldn't use this method anymore.")]
    public void MyThirdDeprecatedMethod()
    {
        Console.WriteLine("Called
MyThirdDeprecatedMethod().");
    }

    // make the program thread safe for COM
    [STAThread]
    static void Main(string[] args)
    {
        BasicAttributeDemo attrDemo = new
BasicAttributeDemo();

        attrDemo.MyFirstDeprecatedMethod();
        attrDemo.MySecondDeprecatedMethod();
        attrDemo.MyThirdDeprecatedMethod();
    }
}
```

همانطور که در [مثال 16-1](#) نیز مشاهده می‌شود، صفت *Obsolete* در فرمهای مختلف مورد استفاده قرار گرفته است. اولین محلی که از این صفت استفاده شده است، متد *MyFirstDeprecatedMethod()* و پس از آن در متد

`MySecondDeprecatedMethod()` است. تنها تفاوت استفاده در این دو حالت آنست که در متد دوم صفت با نام کامل یعنی به همراه کلمه `Attribute` مورد استفاده قرار گرفته است. نتیجه هر دو اعلان یکسان است. همانطور که گفته بودیم، صفتها می‌توانند دارای پارامتر نیز باشند :

```
[Obsolete("You shouldn't use this method anymore.")]  
public void MyThirdDeprecatedMethod()  
...
```

این پارامتر، ویژگی خاصی را به صفت می‌افزاید که آن را با دو اعلان قبلی متمایز می‌نماید. نتیجه هر سه اعلان این صفت در زیر آورده شده است. این پیغامها، پیامهای کامپایلر `C#` هستند که به هنگام کامپایل برنامه تولید شده‌اند.

```
>csc BasicAttributeDemo.cs  
Microsoft (R) Visual C# .NET Compiler version 7.10.2292.4  
for Microsoft (R) .NET Framework version 1.1.4322  
Copyright (C) Microsoft Corporation 2001-2002. All rights  
reserved.
```

```
BasicAttributeDemo.cs(29,3): warning CS0612:  
'BasicAttributeDemo.MyFirstDeprecatedMethod()' is obsolete  
BasicAttributeDemo.cs(30,3): warning CS0612:  
'BasicAttributeDemo.MySecondDeprecatedMethod()' is obsolete  
BasicAttributeDemo.cs(31,3): warning CS0618:  
'BasicAttributeDemo.MyThirdDeprecatedMethod()' is obsolete:  
'You shouldn't use this method anymore.'
```

همانطور که ملاحظه می‌کنید، سومین اعلان صفت در این برنامه که با پارامتر همراه بود، باعث شده است تا پارامتر صفت نیز به عنوان بخشی از پیام نمایش داده شده توسط کامپایلر، نشان داده شود. در مورد دو صفت دیگر نیز مشاهده می‌شود که تنها پیغامی ساده تولید گردیده است.

مثال 16-1 شامل صفت دیگری نیز می‌باشد. این صفت `STAThreadAttribute` است که معمولاً در ابتدای کلیه برنامه‌های `C#` و قبل از آغاز متد `Main()` قرار می‌گیرد. این صفت بیان می‌دارد که برنامه `C#` مورد نظر می‌تواند با کد مدیریت نشده COM از طریق `Simple Threading Apartment` ارتباط برقرار نماید. استفاده از این صفت در هر برنامه‌ای می‌تواند مفید باشد، چراکه شما بعنوان برنامه نویس هیچ‌گاه اطلاع ندارید که آیا کتابخانه ثالثی که از آن استفاده می‌کنید، قصد برقراری ارتباط با COM را دارد یا نه؟ (در صورتیکه با برخی از اصطلاحات بکار رفته آشنایی ندارید اصلاً نگران نشوید. در اینجا هدف تنها نشان دادن موارد استفاده از صفتهاست.)

صفتها می‌توانند دارای چندین پارامتر باشند. در **مثال 16-2**، استفاده از دو پارامتر برای یک صفت نشان داده شده است.

مثال 16-2

```
using System;  
public class AnyClass
```

```
{
    [Obsolete("Don't use Old method, use New method",
true)]
    static void Old( ) { }

    static void New( ) { }

    public static void Main( )
    {
        Old( );
    }
}
```

همانطور که در [مثال 16-2](#) مشاهده می کنید، صفت مورد استفاده دارای دو پارامتر است. پارامتر اول که یک جمله متنی است و همانند [مثال 16-1](#) عمل می کند. پارامتر دوم نیز بیان کننده نوع پیغامی است که این صفت در هنگام کامپایل تولید می کند. در صورتیکه این مقدار برابر با True باشد، بدین معناست که در هنگام کامپایل پیغام خطا تولید می شود و کامپایل برنامه متوقف می گردد. در حالت پیش فرض مقدار این پارامتر برابر با False است که بیان می دارد، به هنگام کامپایل تنها پیغام هشدار تولید خواهد شد. در پیغام این برنامه، عنصری از برنامه را که نباید از آن استفاده شود معین شده و جایگزین آن نیز معرفی می شود.

AnyClass.Old() is obsolete: 'Don't use Old method, use New method'

نکته مهمی که باید در مورد صفتها در نظر بگیرید آنست که اطلاعاتی که توسط صفت در کد برنامه قرار می گیرد، توسط سایر برنامه ها نیز قابل تفسیر و استفاده است.

انواع پارامترهای صفت (پارامترهای *Named* و *Positional*)

همانطور که در بالا نیز اشاره شد، صفتها می توانند دارای پارامتر نیز باشند. این پارامترها به دو دسته تقسیم می شوند. پارامترهای محلی (*positional*) و پارامترهای اسمی (*named*). از پارامترهای *positional* در زمانی استفاده می شود که می خواهیم پارامتر مورد نظر بصورت اجباری مورد استفاده قرار گیرد و البته این مسئله یک قانون نیست! چراکه در مورد صفت *Obsolete*، این صفت دارای یک پارامتر *positional* دیگر با نام *error* و از نوع *int* نیز می باشد که ما آنرا در مثال 16-1 لحاظ نکردیم. همانطور که در مثال 16-2 مشاهده کردید، از این پارامتر *positional* می توان برای ایجاد یک خطا در زمان کامپایل برنامه استفاده نمود.

```
[Obsolete("Don't use Old method, use New method", true)]
static void Old( ) { }
```

تفاوت پارامترهای positional با پارامترهای named در آنست که، پارامترهای named با نامشان مورد استفاده قرار می‌گیرند و همیشه اختیاری هستند. در [مثال 3-16](#) صفت DllImport را مشاهده می‌نمایید که دارای هر دو نوع پارامتر positional و named است.

مثال 3-16

```
using System;
using System.Runtime.InteropServices;

class AttributeParamsDemo
{
    [DllImport("User32.dll", EntryPoint="MessageBox")]
    static extern int MessageDialog(int hWnd, string msg,
    string caption, int msgType);

    [STAThread]
    static void Main(string[] args)
    {
        MessageDialog(0, "MessageDialog Called!",
        "DllImport Demo", 0);
    }
}
```

صفت DllImport در [مثال 3-16](#) دارای یک پارامتر positional ("User32.dll") و یک پارامتر named (EntryPoint="MessageBox") است. پارامترهای named در هر مکانی می‌توانند قرار گیرند و مانند پارامترهای positional دارای محدودیت مکانی نیستند. بدین معنا که چون در پارامترهای named، نام پارامتر مستقیماً مورد استفاده قرار می‌گیرد، محل قرار گیری آن در لیست پارامترهای صفت مهم نیست اما در مورد پارامترهای positional چون اسم پارامتر مورد استفاده قرار نمی‌گیرد، این پارامترها حتماً باید در مکانهای تعیین شده و تعریف شده در لیست پارامترهای صفت قرار گیرند. توجه کنید که چون هدف ما تنها آشنایی با صفتها و نحوه استفاده از آنهاست، درباره پارامترهای مختلف صفت DllImport بحث نخواهیم کرد چراکه پارامترهای این صفت نیاز به آشنایی کامل با Win32 API دارد.

در یک بررسی کلی می‌توان گفت که پارامترهای Positional، پارامترهای سازنده (Constructor) صفت هستند و در هر بار استفاده از صفت باید مورد استفاده قرار گیرند، ولی پارامترهای Named کاملاً اختیاری هستند و همیشه نیازی به استفاده از آنها نمی‌باشد.

Target های صفتها (عناصری که صفتها بر روی آنها اعمال می‌شوند)

صفت‌هایی که تا کنون مشاهده کردید، همگی بر روی متدها اعمال شده بودند. اما عناصر مختلف دیگری در C# وجود دارند که می‌توان صفت‌ها را بر روی آنها اعمال نمود. جدول 1-16 عناصر مختلف زبان C# را که صفت‌ها بر روی آنها اعمال می‌شوند را نشان می‌دهد.

عناصر اعمال شونده	قابل اعمال به
all	به تمامی عناصر قابل اعمال هستند.
assembly	به تمام یک اسمبلی
class	کلاسها
constructor	سازنده‌ها
delegates	Delegate ها
enum	عناصر شمارشی
event	رخدادها
field	فیلدها
interface	واسطها
method	متدها
module	ماژولها (کدهای کامپایل شده‌ای که می‌توانند به عنوان قسمتی از یک اسمبلی در نظر گرفته شوند.)
parameter	پارامترها
property	Property ها
returnvalue	مقادیر بازگشتی
struc	ساختارها

هر چند ممکن است استفاده از این Target ها باعث ایجاد ابهام شوند، اما می‌توان با استفاده از این Target ها معین کرد که صفت دقیقاً به عنصر مورد نظر اعمال شود. یکی از صفت‌هایی که بر روی اسمبلی اعمال می‌شود و باعث ارتباط با CLS می‌گردد، صفت *CLSCompliantAttribute* است. CLS یا همان Common Language Specification امکان برقراری ارتباط بین کلیه زبان‌هایی که تحت .Net کار می‌کنند را فراهم می‌نماید. Target های صفت‌ها با استفاده از اسم Target که بعد از آن کولون قرار می‌گیرد، ایجاد می‌شوند. در مثال 4-16 نحوه استفاده از این صفت نشان داده شده است.

مثال 4-16

```

using System;

[assembly:CLSCompliant(true)]

public class AttributeTargetDemo
{
    public void NonClsCompliantMethod(uint nclsParam)
    {
        Console.WriteLine("Called
NonClsCompliantMethod().");
    }

    [STAThread]
    static void Main(string[] args)
    {
        uint myUInt = 0;

        AttributeTargetDemo tgtDemo = new
AttributeTargetDemo();

        tgtDemo.NonClsCompliantMethod(myUInt);
    }
}

```

با استفاده از Target مورد نظر در اینجا یعنی assembly، این صفت بر روی کل اسمبلی اعمال می‌گردد. کد موجود در مثال 4-16 کامپایل نخواهد شد، زیرا uint در متد *NonClsCompliantMethod()* اعلان شده است. در اینجا در صورتیکه فرم پارامتر صفت CLSCompliant را به false تغییر دهید و یا متد *NonClsCompliantMethod()* را به متدی منطبق با CLS تبدیل کنید (مثلا نوع بازگشتی آنرا int تعریف کنید) آنگاه برنامه کامپایل خواهد شد. (توضیحی که درباره CLS میتوانم بیان کنم اینست که CLS مجموعه‌ای از ویژگیها و خواص Net Framework است که به نحوی بیان می‌دارد، برای اینکه زبانهای مختلف تحت Net بتوانند بدون مشکل با یکدیگر ارتباط برقرار نمایند، لازم است از یک سری از قوانین پیروی کنند، در غیر اینصورت امکان برقراری ارتباط با سایر کدهای نوشته شده تحت زبانهای برنامه‌سازی دیگر را نخواهند داشت. برای مثال، استفاده از نوع uint به دلیل اینکه در زبانهای مختلف میتواند به صورتهای متفاوتی پیاده‌سازی شود و یا وجود نداشته باشد، سازگار با CLS نیست و برای اینکه بخواهیم برنامه‌ای منطبق با CLS داشته باشیم نباید از آن استفاده نماییم.)

نکته قابل توجه در مورد مثال 4-16 آنست که در این مثال صفت CLSCompliant به استفاده از یک Target که همان assembly است، مورد استفاده قرار گرفته است و از اینرو تمامی مشخصات این صفت به کلیه اعضای این اسمبلی اعمال خواهند شد. توجه نمایید که در این مثال علت و موارد استفاده از صفتها مشهودتر است، چراکه همانطور که مشاهده می‌نمایید، با استفاده از یک صفت می‌توانیم کنترلی بر روی کل اسمبلی و برنامه قرار دهیم تا در صورتیکه

می‌خواهیم برنامه ما با سایر زبانهای برنامه‌سازی تحت .Net. ارتباط برقرار کند، از متدهای استاندارد و سازگار با CLS استفاده نماییم که این قابلیت بزرگی را در اختیار ما قرار خواهد داد.

تولید صفتهای شخصی

پس از اینکه با طریقه استفاده از صفتهای موجود در زبان آشنا شدید، حال نوبت به ساخت صفتهای شخصی می‌رسد. برای تولید یک صفت (Attribute) باید یک کلاس ایجاد نماییم و این کلاس باید از System.Attribute مشتق شود. کلاسی که از System.Attribute مشتق می‌شود (چه بطور مستقیم و چه بطور غیر مستقیم) یک کلاس صفت (Attribute Class) است. اعلان کلاس صفت باعث ایجاد صفت جدیدی می‌شود که می‌توان از آن در برنامه استفاده نمود. به مثال 5-16 توجه فرمایید.

مثال 5-16

```
using System;
public class HelpAttribute : Attribute
{
}
```

در این مثال به سادگی یک صفت جدید تولید کرده‌ایم و می‌توانیم از آن استفاده کنیم.

```
[Help()]
public class AnyClass
{
}
```

همانطور که قبلاً نیز گفتیم استفاده از کلمه Attribute به دنبال نام صفت الزامی نیست. صفتی که در اینجا ایجاد کرده‌ایم عملاً کار خاصی برای ما انجام نمی‌دهد پس اندکی در کد آن تغییر ایجاد می‌کنیم تا مفیدتر باشد.

مثال 6-16

```
using System;
public class HelpAttribute : Attribute
{
    public HelpAttribute(String Description_in)
    {
        this.description = Description_in;
    }
    protected String description;
    public String Description
    {
        get
        {
            return this.description;
        }
    }
}
```

```

    }
}
[Help("this is a do-nothing class")]
public class AnyClass
{
}

```

همطور که مشاهده می کنید با اضافه کردن چند خط کد توانستیم این صفت را کارا تر کنیم. با قرار دادن یک property در این صفت، پارامتر این صفت بعنوان پیغام نمایش داده می شود.

تعریف و یا کنترل موارد استفاده از یک صفت

AttributeUsage یکی از کلاسهای از پیش تعریف شده در زبان است که با استفاده از آن می توانیم موارد استفاده از صفتی را که تولید کرده ایم را کنترل کنیم.

این کلاس دارای سه property مختلف است که می توان آنها را به هنگام استفاده صفت شخصی تنظیم نمود و مورد استفاده قرار داد.

ValidOn

با استفاده از این property می توانیم مشخص کنیم که صفت تولید شده توسط ما، بر روی کدام یک از عناصر برنامه قابل اعمال هستند. اطلاعات این عناصر از AttributeTarget گرفته می شود و می توان عناصر مختلف را بوسیله OR بیتی با یکدیگر ترکیب نمود.

AllowMultiple

با استفاده از این property می توان مشخص کرد که آیا می توان از این صفت بیش از یکبار بر روی یک عنصر برنامه استفاده کرد یا نه.

Inherited

با استفاده از این property می توان قوانین ارث بری این صفت را کنترل نمود. با استفاده از این property می توان مشخص کرد که آیا کلاسی که از کلاسی که صفت بر روی آن اعمال شده، ارث بری می کند نیز، صفت بر رویش اعمال می شود یا نه و یا به عبارتی صفت در کلاس مشتق شده نیز مورد ارث بری قرار می گیرد یا نه.

حال با استفاده از موارد گفته شده در بالا، می خواهیم این مطالب را بر روی صفتی که خودمان تولید کردیم اعمال نماییم. مثال 7-16 را بررسی نمایید.

مثال 7-16

```
using System;
```

```
[AttributeUsage(AttributeTargets.Class), AllowMultiple =
false, Inherited = false ]
public class HelpAttribute : Attribute
{
    public HelpAttribute(String Description_in)
    {
        this.description = Description_in;
    }
    protected String description;
    public String Description
    {
        get
        {
            return this.description;
        }
    }
}
```

در ابتدا به AttributeTargets.Class توجه نمایید. این مشخص می‌کند که صفت Help تنها بر روی کلاسها قابل اعمال است و در صورتیکه از آن بر روی عنصری به غیر از کلاس استفاده نماییم خطایی رخ خواهد داد. بنابراین کد زیر، خطایی تولید خواهد کرد :

```
[Help("this is a do-nothing class")]
public class AnyClass
{
    [Help("this is a do-nothing method")] //error
    public void AnyMethod()
    {
    }
}
```

و کد خطای تولید شده بشکل زیر خواهد بود :

```
AnyClass.cs: Attribute 'Help' is not valid on this
declaration type.
It is valid on 'class' declarations only.
```

توجه کنید که با استفاده از AttributeTargets.All به صفت Help این امکان را می‌دهیم تا بر روی تمامی عناصر موجود اعمال شود. لیست کامل عناصر مجاز نیز بشرح زیر است :

- **Assembly,**
- **Module,**
- **Class,**
- **Struct,**
- **Enum,**
- **Constructor,**
- **Method,**

- **Property,**
- **Field,**
- **Event,**
- **Interface,**
- **Parameter,**
- **Delegate,**
- **All = Assembly , Module , Class , Struct , Enum , Constructor , Method , Property , Field , Event , Interface , Parameter , Delegate,**
- **ClassMembers = Class , Struct , Enum , Constructor , Method , Property , Field , Event , Delegate , Interface**

حال به `AllowMultiple = false` توجه نمایید. با استفاده از این کد، به صفت `Help` اجازه می‌دهیم تا تنها یکبار بر روی عنصری از برنامه اعمال شود. پس کد زیر تولید خطا می‌نماید :

```
[Help("this is a do-nothing class")]
[Help("it contains a do-nothing method")]
public class AnyClass
{
    [Help("this is a do-nothing method")]           //error
    public void AnyMethod()
    {
    }
}
```

و کد خطای تولید شده نیز بصورت زیر است :

`AnyClass.cs: Duplicate 'Help' attribute`

در نهایت نیز به بررسی `Inherited` می‌پردازیم. با استفاده از این ویژگی، معین می‌کنیم در صورتیکه کلاس دیگری بخواهد از روی کلاسی که صفت بر روی آن اعمال شده ارث‌بری نماید، آیا این صفت بر روی آن کلاس نیز اعمال شود یا نه. در صورتیکه مقدار این ویژگی برابر با `True` باشد، کلاس مشتق شده نیز از صفت ارث‌بری می‌نماید. برای یک مثال می‌توانیم حالت زیر را در نظر بگیریم :

```
[Help("BaseClass")]
public class Base
{
}

public class Derive : Base
{
}
```

تمامی حالت‌های مختلف ترکیب این سه ویژگی بصورت زیر است :

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited = false )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited = true )
```

استفاده از پارامترهای *Named* و *Positional* در صفت‌های شخصی

همانطور که در قبل نیز اشاره شد، پارامترهای *Positional* پارامترهای سازنده صفت هستند و در هر بار استفاده از صفت باید لحاظ شوند. حال برای بررسی می‌خواهیم پارامترهایی به صفت *Help* خود اضافه نماییم.

مثال 8-16

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
public class HelpAttribute : Attribute
{
    public HelpAttribute(String Description_in)
    {
        this.description = Description_in;
        this.version = "No Version is defined for this
class";
    }
    protected String description;
    public String Description
    {
        get
        {
            return this.description;
        }
    }
    protected String version;
    public String Version
    {
        get
        {
            return this.version;
        }
        //if we ever want our attribute user to set this
property,
        //we must specify set method for it
        set
        {

```



```

        this.verion = value;
    }
}
[Help("This is Class1")]
public class Class1
{
}

[Help("This is Class2", Version = "1.0")]
public class Class2
{
}

[Help("This is Class3", Version = "2.0", Description =
"This is do-nothing class")]
public class Class3
{
}

```

پس از اینکه این صفت را بر روی کلاس Class1 اعمال کردیم و بخواهیم آنرا کامپایل کنیم با پیغام زیر روبرو می‌شویم :

Help.Description : This is Class1
Help.Version : No Version is defined for this class

چون در اینجا هیچ مقداری برای Version در نظر نگرفته‌ایم، با این پیام مواجه شده‌ایم.

حال نتیجه اعمال این صفت را بر روی کلاس دوم بررسی می‌کنیم.

Help.Description : This is Class2
Help.Version : 1.0

برای پارامترهای اختیاری معمولاً از دو سازنده استفاده نمی‌شود و در عوض از پارامترهای Named استفاده می‌گردد. نکته‌ای که باید به آن توجه کنید آنست که برای پارامترهای Named حتماً باید در تعریف property، از متد set نیز استفاده نمایید در غیر اینصورت با پیغام خطای زیر روبرو می‌شوید :

'Version' : Named attribute argument can't be a read only property

بنابراین در صورتیکه این صفت را بر روی کلاس سوم نیز اعمال کنیم با پیغام خطای مشابهی روبرو خواهیم شد. اگر در کلاس Help تغییری کوچکی ایجاد کنیم و به Description نیز متد set را بیفزاییم، با خطا مواجه نخواهیم شد.

Help.Description : This is do-nothing class
Help.Version : 2.0

اتفاقی که در اینجا رخ می‌دهد آنست که در ابتدا سازنده (Constructor) این صفت به همراه پارامترهای Positional آن فراخوانده می‌شوند و سپس متد set برای هر یک از پارامترهای Named فراخوانده می‌شود.

انواع (type) معتبر برای پارامترهای صفت

انواع معتبر برای پارامترهای صفت بشرح زیر می‌باشند :

bool, byte, char, double, float, int, long, short, string, System.Type, object

همچنین می‌توان از enum و یا آرایه‌ای تک بعدی، که عناصر آن یکی از انواع فوق باشد، نیز استفاده نمود.

استفاده از صفتها در زمان اجرا

تا کنون با طریقه ساخت صفتها و چگونگی استفاده و اعمال آنها بر عناصر مختلف برنامه آشنا شدیم. حال نوبت به آن رسیده است تا ببینیم چگونه می‌توان از صفتها در زمان اجرا استفاده نمود. برای جستجوی (query) یک برنامه درباره صفت موجود در آن، به Reflection نیازمندیم. Reflection قابلیت بدست آوردن اطلاعات مربوط به انواع (Types) مختلف در زمان اجرای برنامه است. با استفاده از توابع Reflection موجود در .Net Framework می‌توانیم با جستجو و پیمایش Metadata مربوط به یک اسمبلی، لیست کاملی از کلاسها، انواع و متدهایی را که برای آن اسمبلی خاص تعریف شده‌اند را، بدست آوریم. به مثال 9-16 در این باره توجه نمایید.

مثال 9-16 : استفاده از Reflection

```
using System;
using System.Reflection;
using System.Diagnostics;

//attaching Help attribute to entire assembly
[assembly : Help("This Assembly demonstrates custom
attributes creation and their run-time query.")]

//our custom attribute class
public class HelpAttribute : Attribute
{
    public HelpAttribute(String Description_in)
    {
        //
        // TODO: Add constructor logic here
        this.description = Description_in;
        //
    }
    protected String description;
```

```

    public String Description
    {
        get
        {
            return this.deescription;
        }
    }
}
//attaching Help attribute to our AnyClass
[HelpString("This is a do-nothing Class.")]
public class AnyClass
{
    //attaching Help attribute to our AnyMethod
    [Help("This is a do-nothing Method.")]
    public void AnyMethod()
    {
    }
}
//attaching Help attribute to our AnyInt Field
[Help("This is any Integer.")]
public int AnyInt;
}
class QueryApp
{
    public static void Main()
    {
    }
}

```

مبحث صفتها بسیار گسترده است و می‌توان ساعتها در مورد صفتهای مختلف بحث نمود. اما آنچه مسلم است تمرین مستمر و پیگیری برنامه نویسی در یافتن مواردی که می‌توان با استفاده از صفتها، برنامه‌ای پویاتر ایجاد نمود، مهمترین عامل در فهم و درک کامل مبحث خواهد بود. درباره Reflection نیز در آینده در یان سایت مفصلا توضیح خواهم داد.

خلاصه :

در این درس با صفتها آشنا شدید. یاد گرفتید که چگونه از صفتهای موجود در .Net Framework استفاده کرده و همچنین چگونه صفتهای شخصی و دلخواه خود را تولید نمایید. همچنین با پارامترهای صفتها و انواع آنها و نیز، عناصری که صفتها بر روی آنها اعمال می‌شوند آشنا شدید. در انتهای درس نیز به مختصر درباره Reflection و چگونگی استفاده از صفتها در زمان اجرا صحبت کردیم. امیدوارم مفید واقع شده باشد.

درس هفدهم :انواع شمارشی در C#

در این درس با انواع شمارشی (Enumerator Types) در زبان C# آشنا خواهیم شد. مطالب مورد بررسی در این درس به شرح زیر می‌باشند :

درک و فهم یک نوع شمارشی یا یک enum

ساخت یک نوع شمارشی جدید

چگونگی استفاده از انواع شمارشی

آشنایی با متدهای مختلف موجود در System.Enum

enum فرم خاصی از انواع مقداری (Value Type) است که از System.Enum مشتق شده و امکان پیمایش درون مجموعه‌ای مشخص را با استفاد از اعداد صحیح برای ما فراهم می‌نماید. با استفاده از enum می‌توان مجموعه‌ای از مقادیر ثابت را تعریف نمود که این مقادیر ثابت با استفاده از یک عدد صحیح قابل دسترسی هستند.

استفاده از enum در برنامه‌ها باعث بالا رفتن خوانایی برنامه می‌شود، چراکه با استفاده از آنها می‌توان با مجموعه‌ای از اعداد صحیح ترتیبی (Sequential)، با عناوینی تعریف شده، کار کرد. برای مثال، در اعلان یک enum، ما مجموعه‌ای از نامهای مورد نظر را تعریف می‌نماییم و در برنامه می‌توانیم از این نامها بصورت ترتیبی استفاده نماییم. Enum به هر یک از عناصر موجود در این مجموعه عددی را تخصیص می‌دهد که شروع این عدد می‌تواند توسط برنامه‌نویس نیز معین گردد. سپس با استفاده از نام عناصر موجود در enum و یا با استفاده از اعدادی که به هر یک از این عناصر تخصیص داده شده، می‌توان enum را پیمایش نمود و به عناصر آن دسترسی داشت.

همانطور که گفته شد، enum یک نوع مقداری (Value Type) است، از اینرو ارث‌بری در مورد آن معنایی ندارد. مساوی قرار دادن دو enum نیز، مقادیر یکی را در دیگری کپی می‌کند. همانطور که در این درس، و در منابع دیگر، خواهید یافت، دسترسی به انواع شمارشی در C# با استفاده از دو کلمه enum و Enum امکان پذیر است. در C# نوع شمارشی enum از نوع BCL خود یعنی Enum ارث‌بری می‌کند! با استفاده از enum یک نوع شمارشی جدید تولید می‌شود و با استفاده از Enum، می‌توان به پیاده‌سازی متدهای استاتیک انواع شمارشی پرداخت.

ایجاد یک نوع شمارشی

Net Framework BCL حاوی enum ها و مثالهای متعددی از استفاده آنها می‌باشد. برای مثال هرگاه که از MessageBox بر روی فرمی استفاده می‌شود، می‌توان از MessageBoxIcon که یک نوع شمارشی است استفاده نمود.

علاوه بر انواع شمارشی تعریف شده و موجود در Net Framework، زمانهایی نیز وجود دارند که می‌خواهیم مجموعه‌ای از عناصر را تعریف کرده و از آنها استفاده نماییم. برای دسترسی به عناصری از نوع صحیح، استفاده از enum باعث خوانا شدن برنامه می‌گردد.

نحوه اعلان یک enum در حالت کلی بصورت زیر است:

```
<modifier> enum <enum_name>
{
    // Enumeration list
}
```

در مثال 17-1 که در زیر ملاحظه می کنید، نحوه اعلان و استفاده از enum مشخص شده است.

مثال 17-1: نحوه اعلان یک enum

```
using System;

// declares the enum
public enum Volume
{
    Low,
    Medium,
    High
}

// demonstrates how to use the enum

class EnumSwitch
{
    static void Main()
    {
        // create and initialize
        // instance of enum type
        Volume myVolume = Volume.Medium;

        // make decision based
        // on enum value
        switch (myVolume)
        {
            case Volume.Low:
                Console.WriteLine("The volume has been turned
Down.");
                break;
            case Volume.Medium:
                Console.WriteLine("The volume is in the
middle.");
                break;
            case Volume.High:
                Console.WriteLine("The volume has been turned
up.");
                break;
        }
        Console.ReadLine();
    }
}
```


در مثال 1-17 نمونه‌ای از اعلان یک enum را مشاهده می‌نمایید. همانطور که دیده می‌شود، اعلان یک نوع شمارشی با استفاده از کلمه کلیدی enum صورت گرفته و سپس به دنبال آن نام این مجموعه مشخص می‌شود. درون گروه‌های باز و بسته { } نیز، عناصر نوع شمارشی اعلان می‌گردند.

نوع شمارشی تولید شده در این مثال از نوع Volume است و در متد Main() از آن برای اعلان myVolume استفاده شده است. از آنجائیکه enum یک نوع مقداری است، می‌توانیم بطور مستقیم آنرا مقداردهی نماییم. پس از آنکه متغیر myVolume مقداردهی شد، می‌توان همانند سایر انواع مقداری، مانند int، از آن استفاده نمود. در حلقه switch، متغیر myVolume با عناصر enum مقایسه می‌شوند.

در هربار استفاده از عناصر enum تولید شده، از نام enum تولید شده، در اینجا Volume، در ابتدای نام عنصر استفاده می‌نماییم (Volume.Medium)، زیرا در صورتیکه در یک برنامه چندین enum وجود داشته باشند که دارای عناصری با نامهای یکسان باشند، در صورت عدم استفاده از نام enum مورد نظر قبل از عنصر، ابهام بوجود آمده و دچار مشکل می‌شویم.

بطور پیش فرض، به اولین عنصر تعریف شده در enum مقدار صفر تخصیص داده می‌شود که این مقدار تنها بعنوان اندیسی جهت دسترسی به این عنصر در نظر گرفته می‌شود. سایر عناصر enum نیز بطور صعودی مقدار می‌گیرند و به هر عنصر یک واحد افزوده می‌شود. در مثال 1-17، عنصر Low دارای اندیس صفر بوده و سایر عناصر به ترتیب مقدار 1 و 2 خواهند داشت.

در C#، برای موارد خاص می‌توان مقادیر پیش فرض در نظر گرفته شده برای عناصر enum را تغییر داد. به مثال زیر توجه کنید.

enum Months

```
{  
    jan, feb, mar, apr  
}
```

enum Months

```
{  
    jan = 10, feb = 20, mar = 30, apr=40  
}
```

همانطور که مشاهده می‌شود، در اعلان اول، از مقدار پیش فرض استفاده شده، که در این حالت jan = 0, feb = 1, mar = 2 و apr = 3 خواهند بود. اما در اعلان دوم، برنامه‌نویس بنا به نیاز خود، تشخیص داده تا به هر یک از عناصر enum مقداری دلخواه را نسبت دهد.

هر چند به تمامی عناصر enum مقداری نسبت داده می‌شود ولی از این مقدار نمی‌توان بطور مستقیم در تخصیص دهی مقدار به متغیری دیگر استفاده نمود. به مثال زیر توجه نمایید :

`int x = Months.jan;` // این دستور نادرست است

`int x = (int) Months.jan` // صحیح

برای استفاده از مقدار تخصیص داده شده به عناصر enum، باید از Casting استفاده نماییم. بدین معنا که باید نوع متغیری را که می‌خواهیم مقدار را به آن نسبت دهیم، باید مشخص شود. در مثال فوق `(int) Months.jan` معین می‌کند که مقدار تخصیص داده شده به `jan` به متغیری نسبت داده می‌شود که از نوع `int` است و یا به عبارت صحیح تر، مقدار تخصیص داده شده به عنصر `enum`، در فرمت `int` به متغیر مورد نظر تخصیص داده می‌شود. در ادامه مبحث، توجه شما را به مثالی دیگر درباره `enum` جلب می‌نمایم. توجه نمایید که نکات جدیدی نیز در این مثال گنجانده شده‌اند.

مثال 2-17: ساخت `enum` پایه و تخصیص دهی اعضای آن

```
using System;
```

```
// declares the enum
```

```
public enum Volume : byte
```

```
{
```

```
    Low = 1,  
    Medium,  
    High
```

```
}
```

```
class EnumBaseAndMembers
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // create and initialize
```

```
        // instance of enum type
```

```
        Volume myVolume = Volume.Low;
```

```
        // make decision based
```

```
        // on enum value
```

```
        switch (myVolume)
```

```
        {
```

```
            case Volume.Low:
```

```
                Console.WriteLine("The volume has been  
turned Down.");
```

```
            break;
```

```
            case Volume.Medium:
```

```
                Console.WriteLine("The volume is in the  
middle.");
```

```
            break;
```

```
            case Volume.High:
```

```
                Console.WriteLine("The volume has been  
turned up.");
```

```
            break;
```

```
        }
```

```

        Console.ReadLine();
    }
}

```

با توجه به مثال 2-17 با نحوه تغییر نوع پایه یک enum آشنا می شوید. همانطور که ملاحظه می نمایید، نوع پایه این enum به byte تغییر یافته است. این امر بیان میدارند که تنها مقادیری از نوع byte قابل تخصیص به عناصر enum هستند.

همانطور که قبلاً نیز اشاره شد، مقدار پیش فرض برای اولین عضو enum یعنی Low برابر با صفر است. چون در این مثال مقدار Low را برابر با یک قرار داده ایم، از اینرو مقادیر دو عضو دیگر آن نیز بصورت Middle=2 و High=3 تغییر خواهند یافت.

نکات پیشرفته درباره enum

در زبان C# هر نمونه از enum فضایی معادل با 4 بایت از حافظه را اشغال می نمایند. این مطلب با استفاده از کلمه کلیدی sizeof قابل بررسی لسا. از اینرو از enum میتوان به عنوان یک ساختمان داده مناسب و کارا یاد کرد. نکته بسیار مهم و جالب در مورد enum در زبان برنامه نویسی C# آنست که، برای هر کلاس enum موجود در کد برنامه، مقادیر رشته ای تخصیص داده شده به عناصر enum در یک اسمبلی و بصورت Metadata ذخیره می گردند، از اینرو دسترسی به این مقادیر رشته ای در کد میسر می شود. همچنین می توان از متدهای مختلف مرتبط با enum نیز استفاده نمود. به مثال ساده زیر توجه نمایید :

enum Language

```

{
    CSharp, Mcpp, VBNet, JScript, IL
}
class App
{
    public static void Main()
    {
        Console.WriteLine("Write the number of the selected
Language");
        string[] langAr =
Enum.GetNames(Type.GetType("Language"));
        for(int i=0; i<langAr.Length; i++)
        {
            Console.WriteLine(i + "." + langAr[i]);
        }
        Language
myLang=(Language)Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Your Language of choice is: " +
myLang);
    }
}

```

Overload کردن عملگرها در C#

مطالبی که در این قسمت مورد بررسی قرار خواهند گرفت به شرح زیر می باشند :

Overload کردن عملگرها چیست ؟

درک اینکه چه زمانی از Overload کردن عملگرها استفاده می کنیم.

چگونگی Overload کردن عملگرها

قوانین معمول در Overload کردن عملگرها

در این مبحث می خواهیم درباره Overload کردن عملگرهای زبان C# صحبت کنیم. Overload کردن عملگرها بدین معناست که با استفاده از عملگرهای موجود در زبان C#, عمل دیگری بغیر از عمل در نظر گرفته شده برای آن عملگر را برای آن تعریف نماییم. در این مبحث با این مفهوم به طور کامل آشنا خواهید شد.

نگاهی بر Overload کردن عملگرها

همانطور که می دانید، در C# همانند سایر زبانهای برنامه سازی، عملگرهای متعددی وجود دارند (برای بررسی عملگرها می توانید به "عملگرها در C#" مراجعه نمایید). این عملگرها برای انواع از پیش تعریف شده در زبان C# مورد استفاده قرار می گیرند. اما در موارد مورد نیاز می توان برای انواع تعریف شده توسط کاربر نیز، عملگرهای خاص مورد نظر را تعریف نمود. با استفاده از Overload کردن عملگرها می توان از عملگرهای تعریف شده نیز همانند عملگرهای موجود در زبان استفاده کرد.

برای درک بهتر اهمیت استفاده از Overload کردن عملگرها، فرض کنید می خواهید عملیات ریاضی را بر روی ماتریس ها انجام دهید. برای انجام این کار، مثلاً ماتریسی دو بعدی ایجاد می کنید و از آن استفاده می کنید. اما می دانید که می خواهید از این کد تولید شده در برنامه های دیگر نیز استفاده کنید.

برای این منظور، یک نوع جدید با نام Matrix اعلان می کنید که این نوع جدید می تواند یک کلاس و یا یک struct باشد. (برای بررسی کلاسها به "کلاسها در C#" و برای بررسی ساختارها به "ساختارها در C#" رجوع نمایید). حال که نوع جدیدی با عنوان Matrix را اعلان نموده اید، مسلماً می خواهید از روی آن نمونه هایی تولید کرده و عملیات ریاضی نظیر جمع و ضرب را بر روی آنها اجرا نمایید. برای انجام چنین عملیاتی می توان دو متد Add() و Product() را پیاده سازی نمود و از آنها استفاده نمود. مثلاً شکل استفاده از این متدها بسته به نحوه آنها می تواند به یکی از شکلهای زیر باشد :

```
Matrix result = mat1.Add(mat2); // instance
```

```
Matrix result = Matrix.Add(mat1, mat2); // static
```

```
Matrix result = mat1.DotProduct(mat2).DotProduct(mat3); //  
and so on...
```

و یا هر حالت دیگری که می توانید برای آن در نظر بگیرید. تعریف این چنین متدهایی و استفاده از آنها، دشوار، غیر عادی و دشوار است. اما در صورتیکه بتوان برای جمع از + استفاده نمود، حالتی بسیار مناسب رخ می دهد. حال فرض کنید می توانیم از + به جای عمل جمع ماتریسها و از * برای ضرب ماتریسها استفاده کنیم. در اینصورت سه فرمت بالا به شکل زیر در خواهند آمد :

Matrix result = mat1 + mat2;

Matrix result = mat1 * mat2;

Matrix result = mat1 * mat2 * mat3 * mat4;

همانطور که ملاحظه می کنید، استفاده از چنین فرمتی به مراتب آسانتر از تعریف متدهایی برای اجرای عملیاتی این چنین است. همچنین استفاده از این فرمت جدید در مسایل پیچیده و دارای عملیات زیاد، بسیار ساده تر و مطمئن تر است.

موارد نامناسب استفاده از *Overload* کردن عملگرها

قبل از اینکه بخش پیاده سازی *Overload* کردن عملگرها بپردازیم لازم است تا بیان داریم که استفاده از *Overload* کردن عملگرها در همه موارد کارآیی ندارد و می تواند باعث گمراهی شود. بهترین موارد استفاده از *Overload* کردن عملگرها، مواردی هستند که عملگری که *Overload* می شود، واقعا بر روی نوع مورد نظر تعریف شده باشد و دارای مفهومی حداقل ریاضی باشد. برای مثال در نظر بگیرید پارکینگی داریم که می خواهیم ورود ماشین در آن را شبیه سازی نماییم. در این حالت *Overload* کردن عملگر + برای ورود ماشین به پارکینگ مناسب نیست، چراکه در این مورد عملگر + مفهومی نمی تواند داشته باشد. توجه نمایید که بیشتر از *Overload* کردن عملگرها در مواردی استفاده می شود که به نحوی به مسایل ریاضی مربوط هستند و یا عملگر مورد نظر برای شیء خاص تعریف شده بوده و یا استفاده از این عملگر بر روی شیء، ابهام ایجاد نکند.

عملگرهای *Overload* شونده

همانطور که تا کنون ملاحظه نموده اید، تمامی عملگرهای زبان C# دارای پیاده سازی داخلی هستند که می توان از این عملگرها در هر عبارتی استفاده نمود. اما مواردی نیز وجود داشتند که برای سهولت کار می توانیم عملگر خاصی را بطور مورد نظر خود پیاده سازی نماییم. پس از اینکه عملگری *Overload* شد، پیاده سازی انجام شده توسط کاربر بر پیاده سازی پیش فرض تقدم پیاده کرده و تنها در صورتیکه عملگر *Overload* شده دارای پیاده سازی نباشد، از پیاده سازی از پیش تعریف شده استفاده خواهد شد. همانطور که می دانید در C# دو نوع عملگر وجود دارد. عملگرهای یگانی (Unary) که قابل *Overload* کردن هستند به شرح زیر می باشند :

+ - ! ~ ++ -- true false

عملگرهای باینری قابل *Overload* شدن نیز به شرح زیر می باشند :

+ - * / % & | ^ << >> == != > <
>= <=

توجه نمایید، هر چند عملگرهای true و false هیچ‌گاه بطور صریح بعنوان عملگر در عبارات استفاده نمی‌شوند، اما آنها را نیز عملگر می‌خوانیم چراکه در بسیاری از عبارات منطقی و شرطی از آنها بعنوان عملگر استفاده می‌گردد.

توجه نمایید که تنها عملگرهای مشخص شده در بالا قابلیت Overload شدن را دارند و نمی‌توان سایر عملگرهای زبان C# را Overload نمود. همچنین توجه کنید که Overload کردن یک عملگر باعث Overload شدن ضمنی سایر عملگرهای مرتبط با آن نیز می‌شود. برای مثال Overload کردن عملگر + باعث Overload شدن ضمنی عملگر ترکیبی += نیز می‌شود. البته توجه نمایید که عملگر انتساب یا = هیچ‌گاه Overload نمی‌شود.

نکته دیگری که باید در مورد عملگرهای Overload شده در نظر گرفت آنست که، Overload کردن عملگرها خواصی نظیر حق تقدم و یا شرکت‌پذیری عملگر را تغییر نمی‌دهد. بعنوان مثال عملگر /، عملگری باینری با حق تقدم مشخص و شرکت‌پذیری از چپ است.

پیاده‌سازی عملگر Overload شده

پیاده‌سازی یک عملگر Overload شده تقریباً شبیه به پیاده‌سازی متدی استاتیک است، با این تفاوت که در اینجا لازم است از کلمه کلیدی operator و عملگر مورد نظر استفاده نماییم. در زیر نمونه‌ای از ساختار کلی عملگر ضرب که برای ماتریسها در نظر گرفته بودیم را مشاهده می‌نمایید.

```
public static Matrix operator *(Matrix mat1, Matrix mat2)
{
    // dot product implementation
}
```

همانطور که مشاهده می‌کنید، متد استفاده شده حتماً باید بطور استاتیک تعریف گردد. از کلمه کلیدی operator نیز پس نوعی که می‌خواهیم برای آن عملگری را Overload نماییم، قرار می‌گیرد که در اینجا Matrix نوع مورد نظر ما است. پس از کلمه کلیدی operator، عملگری که می‌خواهیم Overload کنیم را قرار داده و سپس پارامترهایی که عملگر بر روی آنها اعمال می‌شوند را قرار می‌دهیم. در مثال 1-18، نمونه‌ای از Overload کردن عملگرها را مشاهده خواهید کرد.

مثال 1-18: نمونه‌ای از Overload کردن عملگرها

```
using System;

class Matrix3D
{
    public const int DIMSIZE = 3;
    private double[,] matrix = new double[DIMSIZE,
DIMSIZE];
```

```

// امکان تخصیص مقدار را برای فراخواننده فراهم می‌کند.
public double this[int x, int y]
{
    get { return matrix[x, y]; }
    set { matrix[x, y] = value; }
}

// Overload کردن عملگر + برای استفاده بر روی ماتریسها
public static Matrix3D operator +(Matrix3D mat1,
Matrix3D mat2)
{
    Matrix3D newMatrix = new Matrix3D();

    for (int x=0; x < DIMSIZE; x++)
        for (int y=0; y < DIMSIZE; y++)
            newMatrix[x, y] = mat1[x, y] + mat2[x, y];

    return newMatrix;
}

class MatrixTest
{
    // در متد InitMatrix از آن استفاده می‌شود.
    public static Random rand = new Random();

    static void Main()
    {
        Matrix3D mat1 = new Matrix3D();
        Matrix3D mat2 = new Matrix3D();

        // ماتریسها با مقادیر تصادفی مقداردهی می‌شوند.
        InitMatrix(mat1);
        InitMatrix(mat2);

        // ماتریسها در خروجی نمایش داده می‌شوند.
        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);
        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);

        // عمل جمع ماتریسها صورت گرفته و نتیجه محاسبه می‌گردد.
        Matrix3D mat3 = mat1 + mat2;

        Console.WriteLine();
    }
}

```

```

        Console.WriteLine("Matrix 1 + Matrix 2 = ");
        PrintMatrix(mat3);
    }

    // متدی که در آن ماتریسها با مقادیر تصادفی مقداردهی می‌شوند.
    public static void InitMatrix(Matrix3D mat)
    {
        for (int x=0; x < Matrix3D.DIMSIZE; x++)
            for (int y=0; y < Matrix3D.DIMSIZE; y++)
                mat[x, y] = rand.NextDouble();
    }

    // متد چاپ ماتریس در خروجی.
    public static void PrintMatrix(Matrix3D mat)
    {
        Console.WriteLine();
        for (int x=0; x < Matrix3D.DIMSIZE; x++)
        {
            Console.Write("[ ");
            for (int y=0; y < Matrix3D.DIMSIZE; y++)
            {
                // فرمت‌دهی خروجی.
                Console.Write("{0,8:#.000000}", mat[x, y]);

                if ((y+1 % 2) < 3)
                    Console.Write(", ");
            }
            Console.WriteLine(" ]");
        }
        Console.WriteLine();
    }
}

```

در مثال 18-1، عملگر + مورد Overload شدن قرار گرفته است. برای تمرکز بیشتر بر روی کد، قسمت مربوط به Overload شدن عملگر + را در زیر آورده‌ام:

```

public static Matrix3D operator +(Matrix3D mat1, Matrix3D
mat2)
{
    Matrix3D newMatrix = new Matrix3D();

    for (int x=0; x < DIMSIZE; x++)
        for (int y=0; y < DIMSIZE; y++)
            newMatrix[x, y] = mat1[x, y] + mat2[x, y];

    return newMatrix;
}

```

}

عملگر همیشه بطور استاتیک اعلان می‌شود، چراکه متعلق به یک نوع کلی است و مربوط به نمونه‌ای خاص نمی‌باشد. نوع بازگشتی، `Matrix3D` است و تنها چیزی که این متد را از یک متد عادی متمایز می‌نماید استفاده از کلمه کلیدی `operator` و عملگر `+` است. پیاده‌سازی عملگر `Overload` شده باعث ایجاد نمونه‌ای جدید از `Matrix3D` شده و عمل جمع ماتریس را انجام می‌دهد.

نکاتی چند در مورد `Overload` کردن عملگرها

زبان `C#` قوانینی برای `Overload` کردن عملگرها اعمال می‌کند. یکی از این قوانین آنست که عملگر `Overload` شده باید از نوعی که مورد استفاده قرار می‌گیرد، اعلان شود.

قانون بعدی اینست که به هنگام پیاده‌سازی عملگرهای مقایسه‌ای نظیر `>`، `<` و `==`، باید حالت‌های ترکیبی آنها را نیز پیاده‌سازی نمود. برای مثال در صورتیکه عملگر `>` را `Overload` می‌کنید، باید عملگر `> =` را پیاده‌سازی نمایید.

نکته دیگر اینکه، پس از `Overload` کردن عملگرها، عملگرهای ترکیبی آنها نیز قابل استفاده هستند. توجه نمایید که عملگرهای مقایسه‌ای که در بالا اشاره شد از این قاعده مستثنا هستند. همانطور که در قبل نیز اشاره شد، پیاده‌سازی عملگر `+` باعث می‌شود تا بتوان از `= +` نیز استفاده نمود.

پایگاه داده در سی شارپ - بخش اول

پیش گفتار

در این مطلب، که آغاز سری مطالب مرتبط با پایگاه‌های داده و نحوه برقراری ارتباط با آنها است، در ابتدا به معرفی مختصری از پایگاه داده پرداخته می‌شود و سپس مقدمات کار با پایگاه داده آموزش داده شده است. از آنجائیکه برقراری ارتباط با پایگاه داده نیازمند داشتن دانشی از زبان `SQL` است، از اینرو در این سری مطالب، ابتدا به بیان دستورات بنیادی `SQL` پرداخته شده و سپس نحوه استفاده از آنها بیان شده است. برای سادگی کار و جلوگیری از پیچیده شدن مطالب در ابتدا کار، در مطالب نخستین از پایگاه داده `MS Access` استفاده شده است که در بسته نرم افزاری `MS Office` قرار دارد. پس از بررسی مقدمات و آموختن نحوه برقراری ارتباط با یک پایگاه داده، به سراغ مطالب پیشرفته خواهیم رفت و با بسط مطالب و عمیق شدن در زبان `SQL`، نحوه کار با `MS SQL Server 2000` را نیز فرا خواهیم گرفت. در دروس ابتدایی بیشتر سعی شده تا برای یادگیری بهتر و سادگی مطالب، قسمتی از کد نویسی‌های مربوط به برقراری ارتباط با پایگاه داده را کم کرده و این کار را به `Visual Studio IDE` محول سازیم اما در ادامه کار، با تمامی قسمت‌ها و کد نویسی کامل آشنا خواهیم شد.

برای شروع کار، به MS Access (ترجیحا نسخه 2003) و Visual Studio .Net (بهتر است از نسخه 2003 استفاده کنید اما تمامی کدها با نسخه 2002 نیز قابل اجرا هستند اما توجه نمایید که در صورت استفاده از Visual Studio .Net 2005 و یا Visual C# Express Edition برای اجرای برخی کدها دچار مشکل خواهید شد.)

مقدمه

پایگاه داده یا همان **Database** مجموعه ای سازمان داده شده از اطلاعات است که به شکل جداول ذخیره می شوند. برای سازمان دهی این اطلاعات روشهای متفاوتی وجود دارد که هدف تمامی آنها فراهم کردن روشهایی مناسب برای سهولت در برقراری ارتباط با پایگاههای داده و استفاده از اطلاعات موجود در آنها است. سیستم مدیریت پایگاه داده (**DBMS**)، مکانیزمی را جهت ذخیره سازی و بازیابی اطلاعات در پایگاه داده فراهم می نماید. در حقیقت **DBMS** باعث می شود تا برنامه نویس بدون نگرانی درباره چگونگی ذخیره سازی داده ها در پایگاه داده و ساختار آنها، به اطلاعات دسترسی پیدا کند و بتواند داده های جدید را در آن ذخیره نماید.

امروزه اکثر پایگاه داده های محبوب از نوع رابطه ای (**Relational**) هستند. همانند پایگاه داده های عادی، در پایگاه داده های رابطه ای نیز دسترسی به اطلاعات ذخیره شده در جداول از طریق زبان پرس و جوی ساخت یافته یا همان **SQL** میسر میگردد که زبانی استاندارد است و توسط اکثر نرم افزارهای مرتبط با پایگاه داده مورد استفاده قرار می گیرد. از جمله سیستمهای پایگاه داده رابطه ای می توان به **MS SQL Server**، **Oracle™**، **Sybase™**، **DB2™**، **Informix™** و **MySQL™** اشاره کرد.

زبانهای برنامه نویسی از طریق یک **Interface** (یا همان نرم افزاری که ارتباط بین **DBMS** و یک برنامه را فراهم می کند) به پایگاه داده متصل شده و با آنها به تعامل میپردازند. در **C#** برقراری ارتباط با پایگاه داده از طریق **ADO.Net** انجام میشود. **ADO.Net** در حقیقت رابط بین نرم افزار و پایگاه داده است و امکانات ویژه ای را جهت دسترسی به اطلاعات موجود در آن در اختیار برنامه نویس قرار می دهد.

مدل پایگاه داده رابطه ای (**Relational Database Model**)

مدل پایگاه داده رابطه ای، نمایش منطقی ای از داده هاست که رابطه موجود بین داده ها را، بدون درگیر شدن با ساختار فیزیکی آنها، نشان میدهد. پایگاه داده رابطه ای از جداول تشکیل میشود. هر جدول خود از سطرها یا رکوردها (**record/row**) و ستونها یا فیلدها (**columns/fields**) تشکیل میشود. در شکل 1، جدول نمونه ای به نمایش گذاشته شده است که نام آن **Employee** است و هدف آن نمایش دادن اطلاعات پرسنلی کارمندان یک اداره است. این جدول از 6 رکورد تشکیل شده و فیلد **number** متعلق به هر رکورد در آن **Primary Key** است که جهت ارجاع به داده ها در جدول مورد استفاده قرار می گیرد. **Primary Key** فیلد یا فیلدهایی در جدول است (هستند) که شامل داده هایی یکتا باشند، بدین معنا که هیچ رکورد دیگری مشابه با آن ندارد. با استفاده از مفهوم **Primary Key**، این تضمین وجود دارد که هر رکورد حداقل با یک مقدار یکتا قابل شناسایی است. یک مثال خوب برای فیلد **Primary Key** فیلدی است که حاوی کد ملی افراد باشد، چراکه در یک جامعه به ازای هر شخص یک کد ملی یا یک شماره

منحصر بفرد وجود دارد و هیچ دو شماره ملی مشابهی یافت نمی شود. در جدول نمونه ما، فیلد **Primary Key** حاوی شماره پرسنلی کارمندان است. همانطور که مشاهده می شود اطلاعات جدول بر اساس فیلد **number** مرتب شده است. در اینجا، اطلاعات بصورت صعودی مرتب شده اند. (حالت دیگر مرتب شدن حالت نزولی است).

هر ستون از جدول، فیلدی متفاوت را نشان میدهد. معمولاً رکوردها در جدول منحصر بفرد هستند (بوسیله **Primary Key**) اما مقادیر فیلدهای مختلف می تواند مشابه با یکدیگر باشد. برای مثال، 3 رکورد مختلف در فیلد **Department** از جدول **Employee** حاوی مقدار 413 هستند.

number	name	department	salary	location
23603	میشم	413	1000	تهران
24568	علی	413	1500	تهران
34589	محمد	642	1300	مشهد
35761	ناصر	611	1800	اصفهان
47132	مریم	413	1500	تهران
78321	فاطمه	611	2500	اهواز

با توجه به اینکه حجم اطلاعات قابل ذخیره در یک جدول نامحدود است، از اینرو باید بتوان با استفاده از روشی تنها به آن قسمت از اطلاعات دسترسی پیدا کرد که مورد نظر کاربر است. برای این منظور از **SQL** استفاده می نماییم. **SQL** مجموعه دستوراتی را فراهم می نماید که با استفاده از آنها قادر خواهیم بود تا اطلاعات مورد نظر را از پایگاه داده انتخاب (**SELECT**) کرده و مورد استفاده قرار دهیم.

نگاهی بر پایگاه داده رابطه ای

در ادامه مطالب این قسمت، با استفاده از یک پایگاه داده نمونه بنام **Book** با دستورات اولیه **SQL** آشنا خواهیم شد. پایگاه داده مورد نظر ما از چهار جدول تشکیل شده است. این جداول به ترتیب حاوی اطلاعاتی درباره نویسنده کتاب (**Authors**)، ناشر کتاب (**Publishers**)، کد شناسایی نویسنده (**AuthorISBN**) و عنوان کتاب (**Titles**) هستند. جدول **Authors** از سه فیلد تشکیل شده است که عبارتند از شماره اختصاصی هر نویسنده، نام و نام خانوادگی. در جدول زیر مشخصات جدول **Authors** نشان داده شده است.

نام فیلد	توضیحات
authorID	شماره شناسایی نویسنده را در پایگاه داده مشخص مینماید. در پایگاه داده Book این فیلد از نوع int تعریف شده و بصورت فیلدی اعلان شده که بطور خودکار مقدارش یک واحد یک واحد اضافه میشود (Auto-Increment Field). با اضافه شدن هر رکورد به

پایگاه داده مقدار این فیلد یک واحد افزوده میشود که همین امر تضمین میکند که مقدار این فیلد همواره منحصر بفرد خواهد بود.	
نام نویسنده کتاب. (از نوع string)	firstName
نام خانوادگی نویسنده. (از نوع string)	lastName

Authors		
lastName	firstName	authorID
Deitel	Harvey	1
Deitel	Paul	2
Nieto	Tem	3
Steinbuhler	Kate	4
Santry	Sean	5
Lin	Ted	6
Sadhu	Praveen	7
McPhie	David	8
Yaeger	Cheryl	9
Zlatkina	Marina	10
Wiedermann	Ben	11
Liperi	Jonathan	12
Listfield	Jeffrey	13

جدول **Publisher** از دو فیلد تشکیل گردیده است که نمایش دهنده شماره شناسایی ناشر و همچنین نام ناشر است. شکل زیر مشخصات جدول **Publisher** را نشان میدهد.

نام فیلد	توضیحات
publisherID	شماره شناسایی ناشر را در پایگاه داده نشان میدهد. این فیلد که از نوع int است و بصورت خودکار مقدارش افزوده میشود، فیلد Primary Key جدول Publisher نیز میباشد.
publisherName	نام ناشر کتاب. (از نوع string)

Publishers	
publisherName	publisherID
Prentice Hall	1
Prentice Hall PTG	2

جدول **AuthorISBN** شامل دو فیلد است که نشان دهنده شماره شناسایی نویسنده و شماره **ISBN** کتابهایی است که یک نویسنده خاص آنها را نوشته است. با استفاده از این جدول میتوان کتابهایی را که یک نویسنده خاص آنها را نوشته بدست آورد. شکل زیر مشخصات جدول **AuthorISBN** را نشان میدهد.

نام فیلد	توضیحات
authorID	شماره شناسایی نویسنده کتاب را نشان میدهد. با استفاده از این فیلد میتوان کتابهای مرتبط با هر نویسنده را در پایگاه داده پیدا کرد. این فیلد که از نوع int میباشد، در جدول Author نیز قرار دارد.
isbn	شماره ISBN مربوط به هر کتاب. (string)

AuthorISBN	
authorID	ISBN
1	0130125075
2	0130125075
1	0130132497
2	0130132497
1	0130161438
2	0130161438
3	0130161438
1	0130284173
2	0130284173
3	0130284173
6	0130284173
7	0130284173
1	0130284181
2	0130284181
3	0130284181
8	0130284181
1	013028419X
2	013028419X
3	013028419X
1	0130293636
2	0130293636

جدول **Titles** از شش فیلد تشکیل شده است که عبارتند از **ISBN** مربوط به هر کتاب، عنوان کتاب، ویرایش، سال انتشار و شماره شناسایی ناشر، نام عکس جلد هر کتاب و قیمت هر کتاب. در شکل زیر مشخصات این جدول نشان داده شده است.

نام فیلد	توضیحات
isbn	شماره ISBN مربوط به هر کتاب. (string)
title	عنوان کتاب. (string)
editionNumber	شماره ویرایش کتاب (string)
copyright	سال نشر کتاب (int)
publisherID	شماره شناسایی ناشر (از نوع int). مقادیر این فیلد میبایست با شماره ای در جدول Publisher همخوانی داشته باشد.
imageFile	نام فایلی که عکس روی جلد کتاب در آن قرار دارد. (string)
price	قیمت کتاب. (از نوع اعداد حقیقی)

مشاهده جدول Titles

در شکل زیر رابطه بین جداول موجود در پایگاه داده **Book** نشان داده شده است. خط اول در هر جدول بیانگر نام جدول است. فیلدهایی که بصورت **Bold** نشان داده شده اند، بیانگر فیلدهای **Primary Key** هستند. همانطور که قبلاً نیز گفته شد، فیلدهای **Primary Key** نمایش دهنده رکوردهایی منحصر بفرد در جدول هستند، از اینرو هر رکورد در جدول، میبایست دارای مقداری در فیلد **Primary Key** خود باشد و این مقدار میبایست منحصر بفرد باشد. از این اصل، بعنوان قانون "جامعیت موجودیتها" یاد میشود (**Rule Of Entity Integrity**). توجه نمایید که در جدول **AuthorISBN** دو فیلد بعنوان **Primary Key** نشان داده است، که این بدین معناست که در این جدول **Primary Key** مرکب وجود دارد، از اینرو هر رکورد در این جدول میبایست دارای **authorID** و **isbn** منحصر بفردی (بطور مشترک) باشد. برای مثال، ممکن است رکوردهای بسیاری در جدول یافت شوند که مقدار فیلد **authorID** آنها برابر با 2 باشد، اما چون ترکیب **authorID** و **isbn** میبایست منحصر بفرد باشد، از اینرو هیچ دو رکوردی نباید یافت شود که دارای **authorID** و **isbn** مشابه باشد. همچنین ممکن است رکوردهای مختلفی مقدار فیلد **isbn** آنها برابر با 0130895601 باشد، اما تنها یک فیلد یافت میشود که مقدار **authorID** آن برابر با 2 و مقدار **isbn** آن برابر با 0130895601 باشد.

توجه : در صورتیکه مقداری برای فیلد **Primary Key** انتخاب نشود، اصل جامعیت موجودیتها نقض شده، از اینرو **DBMS** خطایی را گزارش میکند.

توجه : وارد کردن مقادیر مشابه برای فیلد **Primary Key** باعث می شود تا **DBMS** خطایی را گزارش نماید.

خطهای رسم شده بین جداول، رابطه (**Relationship**) بین جداول را نشان می دهد. برای مثال خط رسم شده بین جدول **Publisher** و **Titles** را در نظر بگیرید. همانطور که مشاهده میشود، در سمت **Publisher**، بر روی خط عدد 1 قرار دارد و در سمت **Titles** علامت ∞ (بینهایت) قرار گرفته است. خط رسم شده بین این دو جدول، بیانگر یک رابطه "یک به چند" (**One-to-Many**) است که بیان میدارد، به ازای هر ناشر در جدول **Publishers**، تعداد زیادی کتاب می تواند در جدول **Titles** قرار داشته باشد. همچنین توجه داشته باشید که خط رسم شده بین این دو جدول، از فیلد **publisherID** در جدول **Publisher** آغاز شده و به فیلد **publisherID** در جدول **Titles** ختم شده است. فیلد **publisherID** در جدول **Titles** یک **Foreign Key** است، بدین معنا که به ازای هر موجودیت در این جدول، یک مقدار منحصر بفرد در جدولی دیگر وجود دارد و این مقدار منحصر بفرد فیلد **Primary Key** جدول دوم است.

Foreign Key در زمان ایجاد یک جدول مشخص می شود. با استفاده از **Foreign Key**، اصل "جامعیت ارجاع" (**Rule Of Referential Integrity**) مطرح می شود که بیان میدارد، مقدار هر **Foreign Key** باید در فیلد **Primary Key** جدولی دیگر وجود داشته باشد. با استفاده از **Foreign Key** میتوان اطلاعات موجود در جداول مختلف را با یکدیگر "پیوند" (**join**) زد و از آنها استفاده نمود. همواره رابطه ای "یک به چند" بین **Primary Key** و **Foreign Key** وجود دارد، بدین معنا که مقدار فیلد **Foreign Key** می تواند در جدول خودش چندین بار ظاهر شود اما در جدول دیگر تنها می تواند یکبار و آنهم بعنوان **Primary Key** ظاهر شود.

توجه : همانطور که گفته شد، همیشه رابطه ای "یک به چند" از سوی **Primary Key** به سمت **Foreign Key** وجود دارد.

توجه : استفاده از مقداری بعنوان فیلد **Foreign Key** که در **Primary Key** هیچ جدولی وجود نداشته باشد، اصل جامعیت ارجاع را نقض کرده، از اینرو **DBMS** خطایی را گزارش خواهد کرد.

زبان پرس و جوی ساخت یافته (**SQL**)

در این قسمت، به بررسی مختصر زبان SQL خواهیم پرداخت. همانند زبانهای برنامه سازی، زبان SQL نیز از کلمات کلیدی و دستوراتی تشکیل شده است که دستورات کلیدی و مهم آنرا در جدول زیر مشاهده می کنید.

نام دستور SQL	توضیحات
SELECT	فیلدهای مورد نظر را از جدول یا جداولی انتخاب می کند.
FROM	جداولی را مشخص می کند که اطلاعات از آنها باید انتخاب شود و یا قرار است اطلاعات از آنها حذف گردد. در هر دستور SELECT و DELETE وجود دارد.
WHERE	شرایطی را مشخص می کند که تحت آن رکوردهایی خاص انتخاب خواهند شد.
INNER JOIN	رکوردهای متفاوتی را از جداول مختلف به یکدیگر متصل مینماید و مجموعه ای جدید از رکوردها را ایجاد مینماید.
GROUP BY	شرایطی را نشان میدهد که بوسیله آن دسته بندی اطلاعات انجام میشود.
ORDER BY	شرایطی را نشان میدهد که طی آن اطلاعات مرتب می شوند.
INSERT	داده را در جدولی خاص وارد می کند.
UPDATE	داده خاصی را در جدول مورد نظر بروز رسانی می کند. (تغییر میدهد)
DELETE	اطلاعات خاصی را از جدول مورد نظر حذف میکند.

دستور SELECT

یکی از ساده ترین دستورات در SQL، دستوری است که در آن اطلاعات از جدولی خاص انتخاب می شود. چنین عملی با استفاده از دستور SELECT صورت میگیرد و ساده ترین فرمت این دستور بصورت زیر است :

SELECT * FROM *tableName*

که در آن علامت "*" بیان میدارد که تمامی اطلاعات (تمامی ستونها) موجود در جدول انتخاب شده و به نمایش در خواهند آمد و *tableName* نام جدولی است که میخواهیم اطلاعات را از آن انتخاب نماییم.

SELECT * FROM Authors

برای انتخاب فیلد یا فیلدهایی خاص از یک جدول، میتوانیم نام فیلدهای مورد نظر آن جدول را در جلوی دستور SELECT و بجای * قرار دهیم و آنها را با کاما "," از یکدیگر جدا نماییم.

SELECT authorID, lastName FROM Authors

تنها فیلد **authorID** و **lastName** از جدول **Authors** را نشان میدهد.

نکته : در صورتیکه نام فیلد شامل فضای خالی (Space) باشد، در اینصورت میبایست نامو فیلد را داخل دو براکت "[]" قرار دهیم. (SELECT [author ID] FROM Authors)
دستور **WHERE**

در اکثر موارد کاربر در جدول به دنبال اطلاعاتی میگردد که دارای شرایط خاصی است. در این موارد تنها آن رکوردهایی که با شرایط مورد نظر کاربر همخوانی دارند میبایست نمایش داده شوند. SQL با استفاده از دستور **WHERE** در دستور **SELECT**، شرایط مورد نظر را اعمال می نماید. ساده ترین فرم دستور **SELECT** که به همراه **WHERE** باشد، بصورت زیر است :

```
SELECT fieldName1,fieldName2,... FROM tableName  
WHERE criteria
```

که در آن **fieldName** نام فیلدهای مورد نظر، **tableName** نام جدولی که اطلاعات از آن استخراج می شود و **criteria** شرایطی است که بر اساس آن جستجو در جدول صورت میپذیرد. برای مقال، در صورتیکه بخواهیم عناوین کتابهایی را بیابیم که تاریخ نشر آنها بعد از سال 1999 است، از دستور **SELECT** زیر استفاده می کنیم :

```
SELECT title  
FROM Titles  
WHERE copyright > 1999
```

نکته : در زبان **C#** برای اجرای دستورات **SQL** از یک رشته استفاده می کنیم که این رشته حاوی کل دستور (**Query**) مورد نظر ما است. همچنین برای بالا رفتن خوانایی برنامه، دستورات **SQL** را در خطوط جدا مینویسیم.

دستور **WHERE** می تواند حاوی عملگرهای **<**, **<=**, **=**, **>**, **>=** و عملگر **LIKE** باشد. عملگر **LIKE** برای "تطبیق الگو" (**Pattern Matching**) مورد استفاده قرار میگیرد و به همراه ***** و **?** مورد استفاده قرار میگیرد. با استفاده از تطبیق الگو، میتوان به دنبال رشته ای گشت که حاوی الگوی مورد نظر است. برای مثال، **query** زیر، تمامی رکوردهایی از جدول **Authors** را نشان میدهد که **lastName** آنها با کاراکتر **"D"** شروع شده باشند :

```
SELECT * FROM Authors  
WHERE lastName LIKE 'D*'
```

توجه نمایید، استفاده از کاراکتر ***** بیان میدارد که برای تطبیق الگو تنها کفایت تا کاراکتر اول مقدار **D** داشته باشد و سایر کاراکترها هر مقداری می توانند داشته باشند. همچنین اهمیتی ندارد که چه تعداد کاراکتر بعد از کاراکتر **D** در رشته وجود دارد و تنها اهمیت برای ما وجود کاراکتر **D** در ابتدای رشته است.

نکته : تمامی سیستم های پایگاه داده از عبارت **LIKE** پشتیبانی نمیکند.

نکته : در اکثر سیستمهای پایگاه داده، به جای استفاده از کاراکتر "*" در عبارت **LIKE**، از کاراکتر "%" استفاده میشود.

نکته : در برخی از سیستمهای پایگاه داده، کاراکترهای رشته، **Case Sensitive** هستند.

نکته : بهتر است برای تمیز دادن دستورات و عبارات **SQL**، آنها را با حروف بزرگ بنویسیم.

توجه نمایید در رشته ای که مورد تطبیق الگو قرار میگیرد، در صورتیکه کاراکتر "?" قرار گیرد، بدین معناست که بجای کاراکتر "?" تنها یک کاراکتر قرار خواهد گرفت. برای مثال در **query** زیر، تمامی رکورد هایی از جدول **Authors** نمایش داده می شوند که **lastName** آنها با هر کاراکتری شروع شده باشد و به دنبال آن کاراکتر "i" آمده باشد و بعد از آن هر تعداد کاراکتر قرار داشته باشد :

```
SELECT * FROM Authors
WHERE lastName LIKE '?i*'
```

نکته : اکثر پایگاه های داده، به جای استفاده از کاراکتر "?" از کاراکتر "_" در عبارت **LIKE** استفاده میکنند.

نکته : توجه نمایید که پس از عبارت **LIKE**، رشته مورد نظر درون یک جفت ' ' قرار میگیرد.

ORDER BY دستور

نتیجه یک **query** می تواند بصورت صعودی یا نزولی مرتب گردد. این کار با استفاده از دستور **ORDER BY** انجام میشود. ساده ترین فرم این دستور بصورت زیر است :

```
SELECT fieldName1, fieldName2, ... FROM tableName
ORDER BY field ASC
```

```
SELECT fieldName1, fieldName2, ... FROM tableName
ORDER BY field DESC
```

که در **field**، نام فیلدی است که نتایج **query** بر اساس آن مرتب می شود. در صورتیکه بخواهیم نتایج بصورت صعودی مرتب شوند از کلمه **ASC** و در صورتیکه بخواهیم نتایج بصورت نزولی مرتب شوند از کلمه **DESC** استفاده می نماییم. بطور پیش فرض، نتایج **query** بصورت صعودی نمایش داده میشوند.

```
SELECT * FROM Authors
ORDER BY lastName DESC
```

همچنین، با استفاده از **ORDER BY** میتوان نتایج **query** را بر اساس چند فیلد مختلف نیز مرتب نمود :

```
ORDER BY field1 sortingOrder, field2 sortingOrder, ...
```

که در آن *field*، فیلدهایی هستند که مرتب سازی بر اساس آنها صورت میگیرد و *sortingOrder* نوع مرتب سازی هر فیلد را نشان میدهد. توجه نمایید، در مواردی که دو یا چند فیلد برای مرتب سازی در نظر گرفته شده اند، ابتدا مرتب سازی بر اساس فیلد اول انجام می شود و پس از آن مرتب سازی بر اساس فیلدهای دیگر به پیش میرود. در مثال زیر، ابتدا نتایج *query* بر اساس نام خانوادگی (*lastName*) و سپس بر اساس نام (*firstName*) مرتب میشوند :

```
SELECT * FROM Authors
ORDER BY lastName, firstName
```

توجه نمایید که در یک *query* می توان از ترکیب دستورات *WHERE* و *ORDER BY* استفاده نمود :

```
SELECT isbn, title, price FROM Titles
WHERE title LIKE '%How To Program'
ORDER BY title DESC
```

INNER JOIN : ادغام داده ها از جداول مختلف :

طراحان پایگاه داده، معمولا اطلاعات را به جداول مختلفی تقسیم میکنند تا مطمئن باشند که در پایگاه داده اطلاعات بیهوده و زائد ذخیره نمیشود. برای مثال، پایگاه داده *Book* شامل دو جدول *Authors* و *Titles* است. ما با استفاده از جدول *AuthorISBN*، لینکی بین نویسندگان های مختلف و عناوین کتبی که نوشته اند برقرار میکنیم. در صورتیکه این اطلاعات را به جداول مختلف تقسیم نمیکردیم مجبور بودیم تا اطلاعات نویسندگان را برای هر عنوان کتاب در جدول *Titles* در نظر بگیریم و از اینرو یکسری اطلاعات اضافی و تکراری در جدول ذخیره میشد چراکه برای عناوین کتبی که نویسندگان آنها یکسان است، اطلاعات نویسندگان بطور تکراری ذخیره میگردد.

همچنین برای مقاصد آنالیز و تجزیه و تحلیل، ضروری است تا اطلاعات مختلف از جداول مختلف با یکدیگر ادغام شوند و یک مجموعه اطلاعاتی جدید را ایجاد نمایند. با استفاده از عملی تحت عنوان "ادغام جداول"، این عمل قابل اجرا است و بوسیله *INNER JOIN* در دستور *SELECT* آنرا در *SQL* عملی میکنیم. یک *INNER JOIN* رکوردهای مختلفی را از جداول متفاوت با یکدیگر ادغام میکند و این عمل را از طریق تست کردن مقادیر فیلدهایی انجام میدهد که در جداول مورد نظر عمومیت دارند. ساده ترین فرم *INNER JOIN* بشکل زیر است :

```
SELECT fieldName1, fieldName2, ...
FROM table1
INNER JOIN table2
ON table1.fieldName = table2.fieldName
```

که در آن قسمت بعد از عبارت *ON*، فیلدهایی را نشان میدهد از دو جدول با یکدیگر مقایسه میشوند تا معین شود چه رکوردهایی با یکدیگر ادغام میشوند. برای مثال، *query* زیر لیستی از تمام نویسندگان ها به همراه *ISBN* مربوط به کتابهای نوشته شده توسط آنها را ایجاد میکند :

```
SELECT firstName, lastName, isbn
FROM Authors
```

```
INNER JOIN AuthorISBN
ON Authors.authorID = AuthorISBN.authorID
ORDER BY lastName, firstName
```

این **query** فیلدهای **lastName** و **firstName** از جدول **Authors** را با فیلد **isbn** از جدول **AuthorISBN** ادغام میکند و نتیجه را بر اساس فیلدهای **lastName** و **firstName** بصورت صعودی مرتب مینماید. در این **query** به فرمت عبارت نوشته شده بعد از عبارت **ON** که بصورت **tableName.fieldName** است توجه نمایید. در این قسمت مشخص میشود که عمل ادغام بین دو جدول از طریق کدام فیلد انجام خواهد گرفت. استفاده از فرمت نوشتاری **"tableName."** در مواردی ضروری است که نام فیلد در هر دو جدول یکسان باشد.

ادغام اطلاعات از جداول Authors AuthorISBN Titles و Publisher

در اینجا می خواهیم برای پایگاه داده **Book** یک **query** تعریف کنیم که توسط آن عنوان کتاب، شماره **ISBN**، نام نویسنده، نام خانوادگی نویسنده، سال انتشار کتاب و نام ناشر هر کتاب نمایش داده شود. برای کتابهایی که بیش از یک نویسنده دارند، این **query** میبایست رکوردهای مجزایی را برای هر نویسنده نمایش دهد. توجه نمایید که در این **query** نیاز است تا بین هر چهار جدول ادغام صورت گیرد. این **query** در زیر نمایش داده شده است :

```
SELECT    Titles.Title,      Titles.ISBN,      Authors.FirstName,
Authors.LastName, Publishers.PublisherName
FROM
    (
        Publishers
        INNER JOIN Titles
        ON Publishers.PublisherID = Titles.PublisherID
    )
INNER JOIN
    (
        Authors
        INNER JOIN AuthorISBN
        ON Authors.AuthorID = AuthorISBN.AuthorID
    )

ON Titles.ISBN = AuthorISBN.ISBN
ORDER BY Titles.Title;
```

نتیجه این **query** در شکل زیر نمایش داده شده است :

نتیجه query

حال به بررسی این **query** میپردازیم. در خطوط 1 و 2 این **query**، فیلدهای مورد نظر که بعنوان نتیجه در خروجی نمایش داده میشوند، قرار گرفته اند. توجه نمایید که ترتیب قرار گرفتن فیلدها در این لیست، از این جهت اهمیت می یابد که فیلدها در خروجی به همین ترتیب نمایش داده خواهند شد. این **query**، فیلدهای **title** و **isbn** از جدول **Titles**، فیلدهای **lastName** و **firstName** از جدول **Authors**، فیلد **copyright** از جدول **Titles** و فیلد **publisherName** از جدول **Publishers** را انتخاب مینماید. برای اینکه متن این **query** خوانا تر باشد، قبل از نام هر فیلد نام جدول آنرا نیز آورده ایم. (به این روش **Fully Qualified Name** گفته میشود.) در خطهای بعدی، عمل ادغام بین جداول مختلف صورت گرفته است. در این **query** از سه عمل **INNER JOIN** استفاده شده است. باید به نکته توجه شود، اگرچه **INNER JOIN** بر روی دو جدول انجام میشود، اما نتیجه آن می تواند بعنوان ورودی برای **INNER JOIN** دیگری مورد استفاده قرار گیرد چراکه نتیجه یک **INNER JOIN** خود یک جدول است. همچنین برای اولویت بخشیدن به اجرای دستورات از پرانتز استفاده نموده ایم. در ابتدا با دستور زیر آغاز میکنیم:

```
( Publishers INNER JOIN Titles
  ON Publishers.publisherID = Titles.publisherID)
```

که جدول **Publishers** و **Titles** را با استفاده از شرایطی که در آن فیلدها **publisherID** دو جدول با یکدیگر منطبق هستند، ادغام میکند. جدول موقتی که از انجام این **INNER JOIN** تشکیل میشود حاوی اطلاعات کتاب و ناشر مربوط به آن است.

دیگر **INNER JOIN** این **query** عبارت است از:

```
( Authors
  INNER JOIN AuthorISBN
  ON Authors.AuthorID = AuthorISBN.AuthorID )
```

که جدول **Authors** و **AuthorISBN** را در شرایطی که فیلد **AuthorID** با یکدیگر منطبق باشند، ادغام مینماید.

سومین **INNER JOIN** در این **query** عبارت است از:

```
( Publishers
  INNER JOIN Titles
  ON Publishers.PublisherID = Titles.PublisherID
)
INNER JOIN
( Authors
  INNER JOIN AuthorISBN
  ON Authors.AuthorID = AuthorISBN.AuthorID
)
```

```
ON Titles.ISBN = AuthorISBN.ISBN
```

که باعث ادغام دو جدول موقتی ایجاد شده از دو INNER JOIN قبلی، تحت شرایطی میشود که در آن فیلد Titles.isbn برای هر رکورد در اولین جدول موقت با فیلد AuthorISBN.isbn برای هر رکورد از جدول موقت دوم، منطبق باشند. نتیجه نهایی این INNER JOIN نیز، جدول موقتی است که نتیجه مورد نظر را در بر دارد. در انتهای query نیز، نتایج بر اساس فیلد Titles.title مرتب میشوند.

دستور INSERT

این دستور باعث وارد کردن رکوردی جدید در جدول میشود. ساده ترین فرم این دستور بشکل زیر است :

```
INSERT INTO tableName (fieldName1, fieldName2, ... )  
VALUES (value1, value2, ...)
```

که در آن **tableName** جدولی است که در آن رکورد جدید وارد میشود. به دنبال **tableName** لیستی از فیلدها قرار میگیرد و پس از آن عبارت **VALUES** و به دنبال آن لیستی از مقادیر. توجه نمایید که به ازای هر **fieldName** مقدار **value** در جدول وارد میشود. همچنین باید توجه شود که ترتیب و نوع مقادیر وارد شده در **value** میبایست با نام و نوع **field** مورد نظر همخوانی کامل داشته باشد. برای مثال، اگر **fieldName1** بعنوان **firstName** در نظر گرفته شود، آنگاه مقدار **value1** باید رشته ای باشد که درون یک جفت ' ' قرار گرفته و بیانگر نام باشد. بعنوان یک مثال، **query** زیر را در نظر بگیرید :

```
INSERT INTO Authors (lastName, firstName)  
VALUES ('Smith' , 'Sue')
```

این **query** باعث میشود تا رکوردی جدید در جدول **Authors** وارد شود. توجه نمایید در دستور **INSERT** حتما برای فیلدهای **Primary Key** میبایست مقداری وارد گردد، اما در اینجا چون فیلد **Primary Key**، فیلد **authorID** است و این فیلد از نوع **auto-increment** تعریف شده، نیازی به مقداردهی ندارد، چراکه با وارد شده هر رکورد جدید بطور خودکار مقداری برای آن در نظر گرفته میشود. همچنین دقت کنید، بغیر از فیلدهای **Primary Key** در صورتیکه برای فیلدی از یک جدول در دستور **INSERT** مقداری در نظر گرفته نشود، مقدار آن فیلد تهی (Null) در نظر گرفته خواهد شد.

نکته : SQL برای نشان دادن رشته ها از یک جفت ' ' استفاده میکند. برای نشان دادن رشته هایی که خود حاوی ' هستند (مانند O'Reily) باید از فرمت 'O' Reily' استفاده شود.

دستور UPDATE

این دستور باعث ایجاد تغییر در داده های جدول میشود و ساده ترین فرم آن بصورت زیر است :

```
UPDATE tableName  
SET fieldName1 = value1, fieldName2 = value2, ...  
WHERE criteria
```

که در آن *fieldName* هایی که مقادیر آنها تغییر مییابند مشخص شده و مقادیر جدید (*value*) به آنها تخصیص داده میشود. وجود عبارت **WHERE** در دستور **UPDATE** باعث میشود تا رکوردهایی که این فیلدها در آنها تغییر می یابند معین شوند. برای مثال :

```
UPDATE Authors  
SET lastName = 'Jones'  
WHERE lastName = 'Smith' AND firstName = 'Sue'
```

باعث میشود تا رکوردی که در آن **Sue Smith** قرار داشته است، به **Sue Jones** تغییر یابد.

توجه : عدم استفاده از عبارت **WHERE** پس از دستور **UPDATE** ممکن است باعث بروز خطایی منطقی گردد.

DELETE دستور

دستور **DELETE** باعث حذف داده ها از جدول میشود و ساده ترین فرم آن بصورت زیر است :

```
DELETE FROM tableName  
WHERE criteria
```

که در آن **criteria** مشخص کننده رکورد یا رکوردهایی است که از جدول حذف میشوند.

```
DELETE FROM Authors  
WHERE lastName = 'Jones' AND firstName = 'Sue'
```

باعث حذف شدن رکوردی میشود که نام و نام خانوادگی آن **Sue Jones** است.

نکته : عبارت **WHERE** ممکن است با چندین رکورد در جدول مورد نظر همخوانی داشته باشد، از اینرو در دستور **DELETE** دقت نمایید که شرط نوشته شده در **WHERE** تنها با رکورد مورد نظر شما همخوانی داشته باشد تا از حذف شدن ناخواسته اطلاعات جلوگیری شود.

مدل شیء **ADO.NET** (*ADO.Net Object Model*)

مدل شیء ADO.Net فراهم کننده API ای است که بوسیله آن می توان از طریق کد نویسی به پایگاه داده دسترسی پیدا کرد. ADO.Net مخصوص Net Framework. طراحی شده و نسل جدید ADO میباشد. (کلمه ADO مخفف ActiveX Data Objects است.)

در Net Framework، System.Data فضای است که در آن تمامی API های مربوط به ADO.Net قرار دارند. Namespace های پایگاه ADO.Net عبارتند از System.Data.OleDb و System.Data.SqlClient که با استفاده از آنها امکان دسترسی و ایجاد تغییر در Data Source های مختلف فراهم میگردد. (منظور از Data Source هر منبعی داده ایست که میتوان در آن اطلاعات و داده ها را ذخیره نمود.) System.Data.OleDb شامل کلاسهایی است که بوسیله آنها می توان به هر Data Source ای متصل شد، در حالیکه System.Data.SqlClient تنها شامل کلاسهایی جهت استفاده از MS SQL Server 2000 است.

شروع برنامه نویسی با ADO.Net – اتصال به MS Access

برای آغاز کار برنامه نویسی و اتصال به پایگاه داده، با پایگاه داده Access شروع میکنیم که پایگاه داده ای ساده بوده و به همراه بسته نرم افزاری MS Office قابل نصب است. توجه کنید در برنامه زیر، از ساده ترین روش برای اتصال به پایگاه داده استفاده شده و مطالب پیشرفته تر را در آینده بررسی خواهیم نمود.

برای اتصال به پایگاه داده Access میبایست از کلاسهای موجود در System.Data.OleDb استفاده نماییم.

مراحل کار با پایگاه داده بشرح زیر میباشد :

- اتصال به پایگاه داده
- بدست آوردن اطلاعات مورد نظر از پایگاه داده
- نمایش اطلاعات بدست آمده
- قطع ارتباط با پایگاه داده

برای اتصال به پایگاه داده Access از شیء OleDbConnection استفاده میکنیم. توجه کنید که برای اتصال به هر پایگاه داده ای نیاز به یکسری پارامترها و مسیرها است که این اطلاعات در رشته ای تحت عنوان connection string قرار داده میشود و از طریق اطلاعات این رشته به پایگاه داده متصل میشویم. برای اتصال به Access نیز میبایست از یک connection string استفاده نماییم. برای مثال، رشته زیر را در نظر بگیرید :

```
string connectionString =  
    "provider=Microsoft.Jet.OLEDB.4.0;" +  
    "data source=F:\\Samples\\Books.mdb";
```

این رشته، نمونه ای از یک **connection string** جهت برقراری ارتباط با پایگاه داده **Access** است. در قسمت **provider** نام شرکت پشتیبانی کننده پایگاه داده نوشته میشود که برای **Access** این مقدار عبارت است از **Microsoft.Jet.OLEDB.4.0** و در قسمت **data source** محل قرار گرفتن پایگاه داده مشخص میگردد. (محل فیزیکی که در آن پایگاه داده ذخیره شده است).
همچنین به استفاده از **\"\"** جهت نمایش **\"** توجه نمایید.

در مثال زیر، برنامه نمونه ای مطرح شده که در آن نحوه برقراری ارتباط با پایگاه داده **Access** از طریق شیء **OleDbConnection** به نمایش در آمده است.

```
using System;
using System.Data;
using System.Data.OleDb;

class OleDbConnectionAccess
{
    public static void Main()
    {
        // بیان رشته ای که در آن جزئیات ارتباط با پایگاه داده مشخص میشود.
        string connectionString =
            "provider=Microsoft.Jet.OLEDB.4.0;" +
            "data
            source=F:\\Samples\\Books.mdb";

        // ایجاد شیء جدیدی از OleDbConnection جهت برقراری ارتباط با پایگاه
        // که connectionString به سازنده آن ارسال میشود.
        OleDbConnection myOleDbConnection =
            new OleDbConnection(connectionString);

        // ایجاد شیء OleDbCommand
        OleDbCommand myOleDbCommand =
            myOleDbConnection.CreateCommand();

        // قرار دادن ویژگی CommandText از شیء OleDbCommand به دستور SQL
        // که توسط آن یک سطر از جدول Authors انتخاب میشود.
        myOleDbCommand.CommandText =
            "SELECT * "+
            "FROM Authors "+
            "WHERE authorID=1";

        // باز کردن ارتباط بین پایگاه داده با استفاده از متد Open()
        myOleDbConnection.Open();
    }
}
```



```

// ایجاد شیء OleDbDataReader و فراخوانی متد ExecuteReader() از
// SELECT جهت اجرای دستور OleDbCommand شیء
OleDbDataReader myOleDbDataReader =
    myOleDbCommand.ExecuteReader();

// OleDbDataReader خواندن یک سطر از
// Read() با استفاده از متد
myOleDbDataReader.Read();

// نمایش نتیجه و خروجی
Console.WriteLine("myOleDbDataReader[\" firstName\"] =
"+
    myOleDbDataReader["firstName"]);
Console.WriteLine("myOleDbDataReader[\" lastName\"] =
"+
    myOleDbDataReader["lastName"]);
Console.WriteLine("myOleDbDataReader[\" AuthorID\"] =
"+
    myOleDbDataReader["authorID"]);

// بستن شیء OleDbDataReader با استفاده از متد Close()
myOleDbDataReader.Close();

// بستن ارتباط بین پایگاه داده از شیء OleDbConnection
myOleDbConnection.Close();
}
}

```

خروجی این برنامه بشکل زیر است :

```

myOleDbDataReader[" firstName"] = Harvey
myOleDbDataReader[" lastName"] = Deitel
myOleDbDataReader[" AuthorID"] = 1

```

توجه نمایید که در اینجا تنها یک سطر از پایگاه داده فراخوانی شده و با تغییر دستور SQL میتوان نتیجه های دلخواه را بدست آورد.

در اینجا، اولین بخش از سری مطالب مرتبط با پایگاه داده را به اتمام میرسانم. امید است مورد توجه و علاقه دوستان قرار گرفته باشد.

برای دریافت فایل ضمیمه، حاوی پایگاه داده Books و کدهای برنامه این مطلب بر روی لینک زیر کلیک نمایید. برای بازکردن این فایل به برنامه WinRar نیاز دارید.

[Download Attached File](#)

ساختمان داده ها در C# (با رویکردی بر C# 2.0)

این مطلب، آغاز یک سری شش قسمتی از مطالب مهم و کلیدی درباره ساختمان داده و کاربر آن در توسعه و طراحی نرم افزار است. در قسمت اول، مختصری درباره ساختمان داده ها، تعریف ساختمان داده، آنالیز کارایی یک ساختمان داده و بررسی اهمیت این آنالیز صحبت خواهیم نمود. در این بخش همچنین درباره دو ساختمان داده بسیار مهم و رایج در .Net Framework یعنی آرایه ها و لیست ها صحبت خواهیم کرد.

توجه: مطالب موجود در این 6 قسمت، منطبق بر ساختار .Net Framework 2.0 و C# 2.0 تنظیم شده اند. پیشنهاد این مطالب، آشنایی با زبان برنامه سازی C# است. برای فراگیری این زبان می توانید به لینک زیر مراجعه نمایید:

http://csharp-persian.netfirms.com/C_Sharp_List.htm

قسمت اول: مقدمه ای بر ساختمان داده

مطالب مورد بحث در این قسمت بشرح زیر می باشند:

- مقدمه
- آنالیز کارایی یک ساختمان داده
- ساختمان داده خطی، همگون و بسیار متداول: آرایه (Array)
- ایجاد ساختمان داده های کارآ، قابل استفاده مجدد و Type-Safe
- لیست (List): ساختمان داده ای همگون
- نتیجه و جمه بندی

مقدمه

در طول این مطلب، با چندین ساختمان داده موجود در .Net Framework Base Class و همچنین ساختمان داده هایی که توسط کاربر ایجاد می شوند، آشنا خواهید شد. اگر با مفهوم ساختمان داده آشنایی ندارید به تعریف زیر توجه نمایید:

تعریف: ساختمان داده، کلاسی است که جهت سازماندهی داده ها مورد استفاده قرار می گیرد و از عملیات مختلف قابل اجرا بر روی این داده ها، پشتیبانی می نماید.

معمول ترین و آشنا ترین ساختمان داده، آرایه است که شامل مجموعه ای از داده ها است که پشت سر هم قرار گرفته اند و از طریق یک اندیس مشخص قابل دسترسی هستند.

قبل از شروع این قسمت، مطالب مورد بررسی در مجموعه شش قسمتی مربوط به ساختمان داده را مرور می کنیم تا هدف نهایی از این سری مطالب برای شما مشخص گردد.

در قسمت اول، نگاهی بر اهمیت ساختمان داده ها و چگونگی تاثیر آنها بر کارایی یک الگوریتم خواهیم کرد. برای تعیین تاثیر کارایی یک ساختمان داده بر روی یک الگوریتم، لازم است تا کلیه عملیات موجود در ساختمان داده به دقت مورد بررسی قرار گیرند. در ابتدا تمرکز خود را بر روی دو ساختمان داده آرایه و لیست جمع می کنیم، چراکه با این دو ساختمان داده از قبل آشنایی داشته و می توانیم راحت تر به آنالیز آنها پردازیم. برای آنالیز کارایی آنها، به بررسی عملیات موجود در هر یک از آنها و کارایی هر یک از این عملیات خواهیم پرداخت.

در قسمت دوم، به بررسی صف (Queue) و پشته (Stack) خواهیم پرداخت. همانند لیست ها، صف و پشته نیز مجموعه ای از داده ها هستند و این دو ساختمان داده در Net Framework Base Class Library وجود دارند. برخلاف لیست، که اعضای آن به هر ترتیبی قابل دسترسی هستند، دسترسی به اعضای صف و پشته تنها از طریق ترتیبی خاص قابل دسترسی است. سپس به بررسی چند برنامه و طریقه پیاده سازی صف و پشته خواهیم پرداخت و پس از آن Hashtable ها را مورد بررسی قرار می دهیم. Hashtable ها امکان دسترسی به عناصر را بصورت مستقیم فراهم می نمایند (همانند آرایه یا ArrayList ها) ولی اندیس مورد استفاده در آن کلیدهایی رشته ای هستند. (String Keys)

هر چند ساختمان داده هایی مانند آرایه و لیست، برای دسترسی مستقیم به داده ها در زمانیکه با حجم زیادی از داده ها سر و کار داریم بسیار مفید هستند، این ساختمان داده ها برای جستجو درون داده ها، از بهینگی کمتری برخوردارند. در قسمت سوم، به بررسی ساختمان داده درخت جستجوی دودویی (Binary Search Tree) می پردازیم که روشی مناسب و بهینه برای جستجوی داده های موجود در یک مجموعه است.

اگرچه استفاده از درخت جستجوی دودویی باعث کاهش زمان جستجو می شود، اما دارای نواقص و کاستی هایی نیز می باشد. در قسمت چهارم، به بررسی SkipList ها می پردازیم که ترکیبی از درختهای دودویی و لیست های پیوندی (Linked List) است.

در قسمت پنجم، به بررسی ساختمان داده هایی پرداخته می شوند که می توانند نشان دهنده گراف (Graph) باشند. یک گراف مجموعه ای از گره ها (Nodes) است که هر یک از آنها با لبه هایی (Edge) به گره های دیگر متصل شده اند. برای مثال، نقشه شهرها را میتوان با یک گراف پیاده سازی کرد که در آن شهرها همان گره ها و راه ها و اتوبانهای بین شهرها نشان دهنده Edge ها می باشند. بسیاری از مسایل دنیای واقعی با استفاده از مفهوم گرافها بطور انتزاعی قابل پیاده سازی هستند.

نهایتاً در قسمت ششم، به بررسی ساختمان داده هایی می پردازیم که نشان دهنده Sets و Disjoined Sets هستند. Set، مجموعه ای است از داده ها که بدون هیچ گونه ترتیبی در کنار یکدیگر قرار گرفته اند. Disjoined Set، مجموعه ای از Set هاست که هیچ عنصر مشترکی با یکدیگر ندارند. این دو مفهوم در پیاده سازی برنامه های امروزه کارآیی زیادی دارند و در قسمت ششم به بررسی و نحوه استفاده از آنها خواهیم پرداخت.

آنالیز کارآیی ساختمان های داده

هنگامیکه با یک مسئله برنامه نویسی مواجه می شویم، اغلب برنامه نویسان و طراحان نرم افزار به دنبال پیدا کردن و طراحی الگوریتمی هستند تا بتوانند بواسطه آن کارآیی برنامه خود را افزایش داده و نیازهای کاربر را به بهترین شکل ممکن برآورده نمایند. از دید یک کاربر نرم افزار نیز مفید بودن برنامه اهمیت دارد و کمتر کسی پیدا می شود که به الگوریتم پشت پرده مورد استفاده در برنامه اهمیت دهد. اما استفاده از یک الگوریتم قوی و کارآ در زمینه برنامه یا نرم افزاری که تولید می شود، می تواند نقش بسیار مهمی در افزایش کارآیی برنامه داشته باشد. بعنوان مثال، جستجو درون یک آرایه را در نظر بگیرید. با استفاده یک الگوریتم جستجوی ساده و عادی، به تعداد اعضای موجود در آرایه باید عمل جستجو انجام گیرد. با استفاده از درخت جستجوی دودویی یا SkipList ها، مدت زمان جستجو برای یک عنصر نسبتی لگاریتمی با تعداد اعضای موجود در آرایه پیدا می کند و باعث کاهش زمان جستجو می شود. زمانیکه در انبوهی از داده های بخواهیم به جستجو پردازیم، نوع الگوریتم و ساختمان داده مورد استفاده بسیار می تواند در زمان انجام جستجو موثر باشد، بطوری که می توان اختلاف آنرا بر حسب ثانیه و حتی دقیقه حس نمود.

اگرچه ساختمان داده مورد استفاده در یک الگوریتم به شدت می تواند کارآیی آن را تحت تاثیر قرار دهد، اما روشهای دقیق و مناسبی نیز برای مقایسه و بدست آوردن این کارآیی باید وجود داشته باشند. مسئله ای که برای ما بعنوان یک طراح نرم افزار یا برنامه نویس می تواند اهمیت داشته باشد، کارآیی ساختمان های داده مختلف به هنگام افزایش حجم داده ها است. و این بدان معناست که برای هر عنصر جدیدی که وارد ساختمان داده ما می شود، زمان کار الگوریتم ما به ه شکلی تحت تاثیر قرار می گیرد.

به مثال زیر توجه کنید، فرض کنید باید برنامه ای بنویسید که آرایه ای از رشته ها را بعنوان ورودی دریافت نماید که این آرایه حاوی نام تعدادی فایل است. وظیفه برنامه شما اینست که درون این آرایه به دنبال فایلی با پسوند خاصی بگردد. نمونه ساده این برنامه می تواند بشک زیر باشد :

```
public bool DoesExtensionExist(string [] fileNames, string extension)
{
    int i = 0;
    for (i = 0; i < fileNames.Length; i++)
        if (String.Compare(Path.GetExtension(fileNames[i]), extension, true) ==
0)
            return true;

    return false; // If we reach here, we didn't find the extension
}
```

به نگاهی دقیق تر به این برنامه، متوجه می شوید که در بدترین حالت، یعنی زمانی که فایل مورد نظر در آرایه وجود نداشته باشد و یا وجود داشته باشد اما در آخرین خانه آرایه جای داشته باشد، باید تمامی عناصر آرایه را یکبار مورد بررسی و جستجو قرار دهیم. برای آنالیز مسائل، بطور مثال برای آنالیز مرتب سازی عناصر آرایه (Sort)، به شکل زیر باید فکر کنیم : " فرص می کنم آرایه ای با n عنصر دارم، اگر عنصر دیگری به این آرایه اضافه کنم $n+1$ عنصر خواهم داشت، در این حالت زمان اجرای برنامه من چقدر خواهد شد؟ ". توجه نمایید که کلمه "زمان اجرا" یا Running Time عملا به معنا محاسبه دقیق زمان اجرا برنامه نیست، بلکه محاسبه تعداد مراحل و زمان اجرای این مراحل برای یک عمل خواسته شده است. برای مثال در مورد آرایه، تعداد مراحل به تعداد دفعات دسترسی به عناصر آرایه است. برای جستجو درون یک آرایه، در صورتیکه $n+1$ عنصر در آرایه داشته باشیم، باید $n+1$ بار مقدار مورد نظر خود را با عناصر آرایه مقایسه نماییم. از اینرو می گوئیم مدت زمان جستجو درون یک آرایه بطور خطی با تعداد عناصر موجود در آرایه رابطه مستقیم دارد.

این روش آنالیز که در اینجا مطرح گردید، به روش آنالیز آسیمپوتیک (Asymptotic Analyze) معروف است. روش علامتگذاری که در این روش آنالیز مورد استفاده قرار می گیرد، به روش نشانه گذاری (Big-Oh O Notation) معروف است. برای نشان دادن کارایی جستجو در آرایه غیر مرتب با استفاده از این روش نشانه گذاری، بشکل زیر عمل می شود $O(n)$ که نشان دهنده رابطه زمان با تعداد اعضا است. بدین ترتیب با استفاده از $O(n)$ برای نشان دادن کارایی یک ساختمان داده، این مفهوم بدست می آید که رابطه ای خطی و مستقیم بین افزایش تعداد اعضای از n به $n+1$ و زمان انجام عملیات وجود دارد. بدین ترتیب علامت O نشان دهنده روش نشانه گذاری و روش آنالیز کارایی است و n نشان دهنده چگونگی افزایش مراحل پردازش عمل بسته به تعداد عناصری است که می خواهند مورد پردازش قرار گیرند.

برای محاسبه زمان اجرای یک قطعه کد با استفاده از O-Notation یا همان Asymptotic Analyze می توان از چند مرحله ساده زیر استفاده نمود :

- 1- بدست آوردن مراحل که زمان اجرای یک الگوریتم را شکل می دهند. همانطور که قبلاً ذکر شد، این مراحل برای یک آرایه، خواندن و نوشتن یا همان دسترسی به عناصر آرایه است و این مراحل برای ساختمان های داده ای دیگر متفاوت خواهد بود. باید توجه داشته باشید که برای آنالیز کارایی یک ساختمان داده، تنها به مراحل که توسط خود ساختمان داده ی مورد نظر انجام می شود توجه می گردد و عملیات ساده ای که توسط کامپیوتر انجام می شوند را در این محاسبه به حساب نمی آوریم. برای مثال در کد بالا، برای محاسبه کارایی تنها به تعداد دفعات لازم برای دسترسی به عناصر آرایه توجه کرده و به زمان های لازم جهت ایجاد متغیرها و با زمان محاسبه تساوی دو رشته اهمیت نداده ایم.
- 2- خطهایی از برنامه که در الگوریتم شما تاثیر دارند و می خواهید کارایی را برای آنها محاسبه کنید را پیدا کرده و در کنار آنها عدد 1 قرار دهید.
- 3- اگر این خطوط که آنها را با عدد 1 مشخص کرده اید خود یک حلقه تکرار هستند، بجای عدد 1 در کنار آنها عدد معادل بیشترین تعداد تکرار حلقه را قرار دهید. اگر در جایی حلقه تو در تو دارید، باید از ضرب بیشترین تعداد دفعات تکرار حلقه ها استفاده کنید.
- 4- بزرگترین عدد نوشته شده را پیدا کنید. این عدد زمان اجراست.

حال برای روشن تر شدن موضوع همین مراحل را برای کد نوشته شده در بالا اجرا میکنیم. در کد نوشته شده در بالا، مراحل مورد نظر برای ما، تعداد دفعات دسترسی به عناصر آرایه است. با توجه به کد و با توجه به مرحله 2، تنها در یک خط کد، دسترسی به عناصر آرایه وجود دارد، پس عدد 1 را در کنار این خط قرار می دهیم. با توجه به مرحله 3، این خط در یک حلقه تکرار قرار دارد و این حلقه n با تکرار می شود که n تعداد عناصر آرایه است. طبق مرحله 3، عدد 1 را پاک کرده و بجای آن n در کنار این خط قرار می دهیم. N بزرگترین مقدار نوشته شده است از اینرو زمان اجرا $O(n)$ است.

$O(n)$ یکی از صدها زمان اجرای موجود در آنالیز Asymptotic است. برخی دیگر از زمانهای اجرا عبارتند از $O(\log_2 n)$, $O(n \log_2 n)$, $O(n^2)$, $O(2^n)$ و بدون اینکه وارد جزئیات پیچیدگیهای ریاضیاتی Notation_O شویم، بیان میداریم که هرچه مقدار داخل پرانتز در این آنالیز کوچکتر باشد، عملیات بر روی ساختمان داده کارآمدتر است. برای مثال ساختمان داده ای که زمان اجرای آن $O(\log n)$ است به مراتب کارآمدتر از ساختمان داده ایست که زمان اجرای آن $O(n)$ است. چراکه $\log n < n$

نکته : برای یادآوری باید بگویم که $\log_a b = y$ روش دیگری برای نمایش $a^y = b$ است. از اینرو $\log_2 n$ رشد کندتری نسبت به n دارد، چراکه اگر $n=8$ فرض کنیم، $\log_2 n = 3$ می شود در حالیکه $n=8$ است. در قسمت سوم از این سری مطالب، با درخت های جستجوی دودویی آشنا می شویم که زمان اجرایی آنها $O(\log_2 n)$ است.

زمان اجرای Asymptotic و الگوریتمهای دنیای واقعی

زمان اجرای Asymptotic روشی برای محاسبه کارایی الگوریتم در زمانی است که تعداد عناصر مورد نظر آن به سمت بینهایت میل می کند. هنگامیکه گفته می شود زمان اجرایی یک الگوریتم از الگوریتم دیگر بیشتر است، از لحاظ ریاضی بدین معناست که مراحل اجرایی و زمان اجرایی این مراحل در الگوریتم زمان بر بیشتر از الگوریتمی است که زمان اجرایی کمتری دارد. اما باید توجه کرد که در مواردی، الگوریتم هایی با مراحل اجرایی کم، با اینکه دارای زمان اجرایی بالایی هستند، از الگوریتمهایی که زمان اجرایی پائین تری دارند، سریعتر اجرا می شوند. همانطور که گفته شد این امر معمولاً در مواردی صادق است که الگوریتمی که زمان اجرایی آن بالاست، دارای مراحل بسیار کمی باشد.

بعنوان مثال، الگوریتمهای بسیار زیاد و متفاوتی برای مرتب سازی آرایه ها وجود دارد و هر یک از آنها دارای زمان اجرایی متفاوتی هستند. یکی از ساده ترین و بدیهی ترین الگوریتمهای مرتب سازی (Sort) الگوریتم مرتب سازی حبابی یا Bubble Sort است. زمان اجرایی این الگوریتم $O(n^2)$ است که این مقدار خاطر وجود دو حلقه تودرتو در این الگوریتم بوجود آمده است. در مقابل الگوریتم دیگری به نام Merge Sort وجود دارد که زمان اجرایی آن $O(n \log_2 n)$ است. زمان اجرایی Merge Sort به مراتب کمتر از زمان اجرایی Bubble Sort است. اما برای آرایه هایی با تعداد عناصر کم، زمان اجرایی Bubble Sort کمتر از زمان اجرای Merge Sort است! از آنجائیکه Merge Sort از یک ساختار بازگشتی استفاده میکند و نیز نیم آرایه Sort شده را در انتها با یکدیگر ادغام می نماید، در مورد آرایه های کوچک کارایی چندانی ندارد و چون Bubble Sort تنها از جابجا کردن عناصر مجاور آرایه استفاده می کند، از اینرو برای آرایه ها کوچک الگوریتمی مناسب تر است. (لطفاً در صورتیکه در مورد الگوریتمهای Sort سوال و یا مشکلی دارید حتماً به من ایمیل بزنید) نهایتاً، مراحل کاری Merge Sort از Bubble Sort کمتر است اما از آنجائیکه وزن این مراحل بالاست (هر مرحله احتیاج به زمان پردازشی بالایی دارد) از اینرو برای آرایه هایی عظیم مفید است. حال آنکه برای آرایه هایی با حجم کم الگوریتم Bubble Sort مناسب تر است. پس علاوه بر توجه به زمان اجرایی یک الگوریتم، توجه به موارد استفاده و حجم داده مورد نظر نیز موثر است.

آرایه : ساختمان داده ای خطی، همگون، با امکان دسترسی مستقیم

آرایه ها یکی از ساده ترین و پر استفاده ترین ساختمانهای داده هستند که در تمامی زبانهای برنامه سازی دارای ویژگیهای مشترکی هستند :

- محتویات آرایه در یک فضای حافظه پیوسته ذخیره می شود
- تمامی عناصر یک آرایه باید از یک نوع یا از یک نوع مشتق شده باشند. از اینرو به آرایه، ساختمان داده ای همگون گفته میشود.
- دسترسی به عناصر آرایه بصورت مستقیم و از طریق اندیس است.

عملیات متدوالی که بر روی آرایه ها انجام میشود نیز بشرح زیر است :

تخصیص

دسترسی

در زبان C# هنگامیکه آرایه ای ایجاد میشود، مقدار پیش فرض تهی (Null) برای آن در نظر گرفته میشود. برای مثال خط زیر آرایه تهی BoolArray را ایجاد میکند :

```
bool[] BoolArray;
```

قبل از اینکه بتوانیم از آرایه ها استفاده کنیم، می بایست نمونه ای از آن را ایجاد نماییم تا بتوانیم عناصر مورد نظر را در آن ذخیره کنیم :

```
arrayName = new arrayType[allocationSize];
```

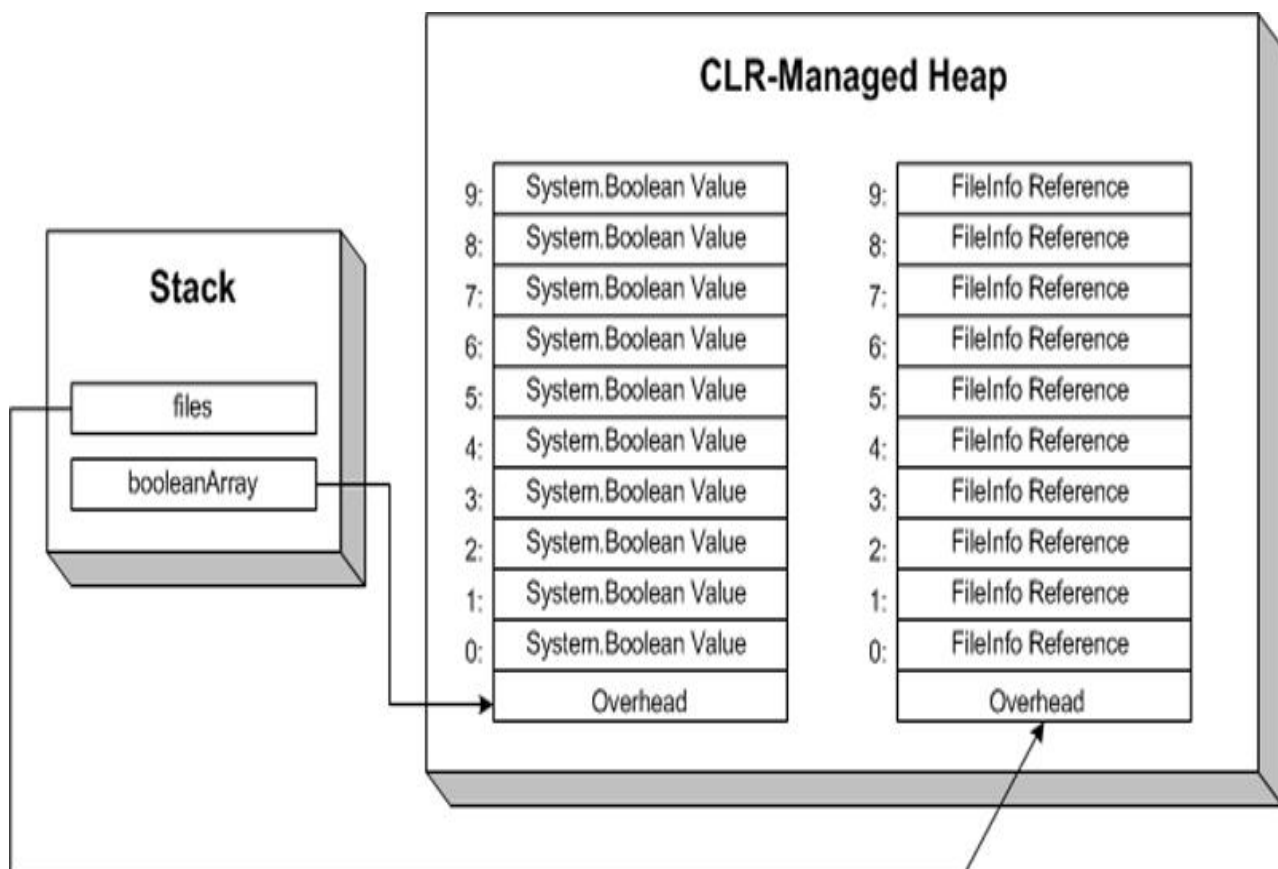
این خط از کد، باعث تخصیص فضای پیوسته ای از حافظه در CLR-Managed Heap به اندازه تعداد allocationSize از arrayType میشود. (برای مثال اگر فضای مورد نیاز برای ذخیره سازی متغیری از نوع int برابر 2 بایت باشد، برای آرایه ای که allocationSize آن 5 است، به 10 بایت حافظه نیاز است.) اگر arrayType از انواع مقداری (Value Type) باشد، آنگاه مقادیری از نوع arrayType و به تعداد allocationSize ایجاد میشود. (این مقادیر ایجاد شده unboxed هستند.) در صورتیکه arrayType از انواع مرجعی (Reference Type) باشند، آنگاه مرجع هایی از نوع arrayType و به تعداد allocationSize ایجاد میگردند. (در صورتیکه تفاوت بین انواع مرجعی و مقداری و فرق بین پشته و heap را نمیدانید، به درک انواع رایج در .Net. مراجعه نمایید.)

برای درک چگونگی نگهداری عناصر آرایه توسط .Net Framework ، به مثال زیر توجه نمایید :

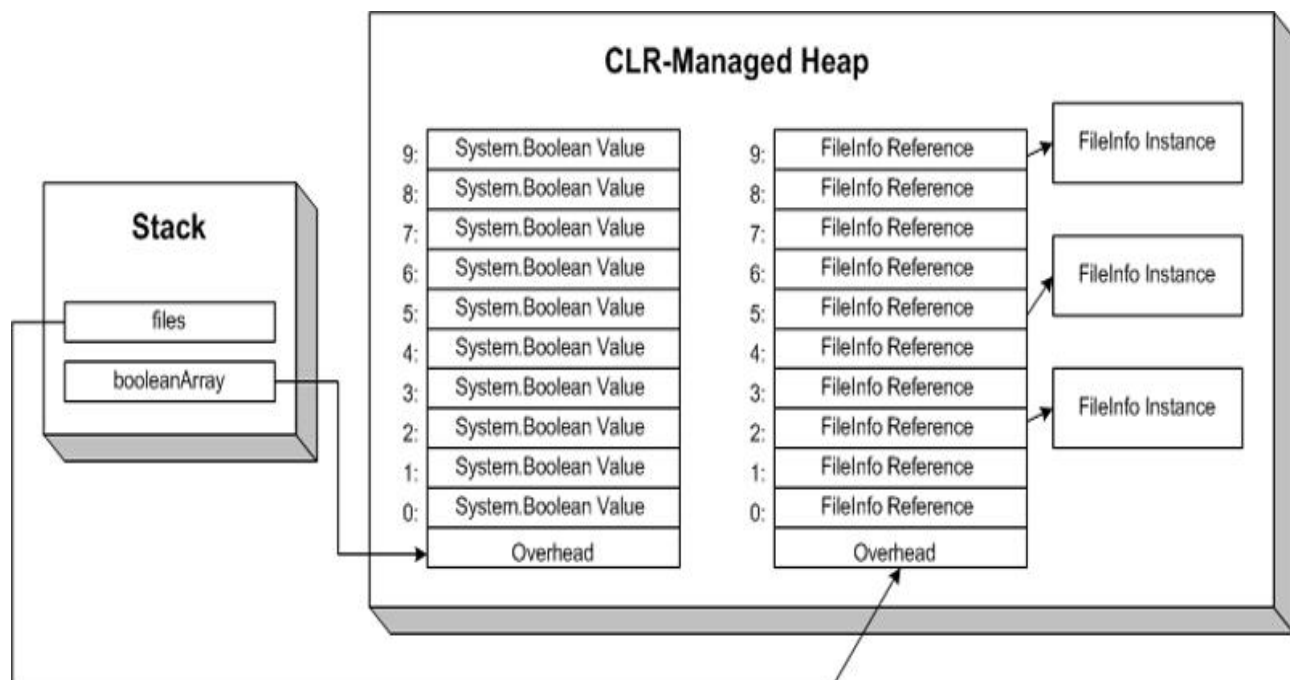
```
bool [] booleanArray;  
FileInfo [] files;
```

```
booleanArray = new bool[10];  
files = new FileInfo[10];
```

در اینجا، آرایه booleanArray از نوع System.Boolean است در حالیکه آرایه files از نوع مرجعی System.IO.FileInfo میباشد. شکل زیر وضعیت CLR-Managed Heap را پس از اجرای این چند کد به نمایش در می آورد :



باید به این نکته نیز توجه نمایید که 10 عنصر موجود در آرایه **Files**، مرجع هایی به نمونه هایی از **FileInfo** هستند.



در **.Net**، تمامی عناصر آرایه امکان نوشته شدن و نوشتن بر آرایه را دارند. نحوه دسترسی به عناصر آرایه به فرم زیر است :

```
// Read an array element
```

```
bool b = booleanArray[7];  
// Write to an array element  
booleanArray[0] = false;
```

زمان اجرایی دسترسی به یک آرایه را بصورت $O(1)$ نشان می دهیم، زیرا زمانی ثابت است. این به معنی آنست که، هر چقدر هم که عناصر آرایه زیاد باشند، زمان دسترسی به یک عنصر یکسان و ثابت است. علت ثابت بودن این زمان آنست که، عناصر آرایه بصورت متوالی در خانه های حافظه قرار می گیرند و به هنگام دسترسی به یک عنصر تنها کافی است تا نقطه شروع آرایه، طول عناصر آرایه و اندیس عنصر مورد نظر را داشته باشیم. در این حالت دسترسی به هر یک از عناصر آرایه زمانی واحد و ثابت را نیاز دارد.

نکته قابل توجه دیگر آنست که، در کدهای مدیریت شده، زمان دسترسی به عناصر آرایه اندکی با آنچه گفته شد متفاوت است. برای مثال در .Net، با هر درخواست برای دسترسی به عنصر یک آرایه، CLR اندیس مورد نظر را نیز چک میکند تا این اندیس درون آرایه قرار داشته باشد. در صورتیکه اندیس مورد نظر خارج از آرایه باشد، استثنای `IndexOutOfRangeException` ایجاد میشود. (برای مثال اگر آرایه ای با 10 عنصر داشته باشیم و اندیس دسترسی به عنصر آرایه را 15 در نظر بگیریم، این استثناء رخ خواهد داد، چراکه اندیس مورد نظر خارج از مرزهای (Bound) آرایه است.) این کنترل باعث می شود تا به هنگام استفاده از آرایه ها مطمئن باشیم که به اشتباه وارد قسمت دیگری از حافظه نشده ایم. (برای مثال در برخی از نسخه های کامپایلر C این کنترل صورت نمی گیرد و شما می توانید به عنصر یازدهم یک آرایه ده عنصری دسترسی پیدا کنید !!! در این حالت اتفاقی که رخ می دهد آنست که، کامپایلر به اشتباه محتویات خانه ای از حافظه را به شما نشان میدهد که اصلا جزو آرایه مورد نظر نیست. این امر باعث بروز اشکالات فراوانی برای برنامه نویس به هنگام تست برنامه میشود. چراکه محتویات خانه ای از حافظه که به اشتباه اندیس دهی شده است، ممکن است تاثیرات نامطلوب و غیر قابل پیش بینی بر روی نتیجه برنامه داشته باشد و بررسی و کنترل منبع این خطا بسیار دشوار است.) توجه نمایید که انجام این کنترل باعث تغییر زمان اجرایی دسترسی به عناصر آرایه $O(1)$ نمیشود چراکه با افزایش تعداد عناصر آرایه این زمان تغییر نمیکند.

نکته : انجام چنین کنترلی بر روی اندیس دسترسی به عناصر آرایه، باعث افزایش کارایی در برنامه هایی میشود که بطور عظیمی با آرایه ها و دسترسی به عناصر آنها سروکار دارند. در کدهای مدیریت نشده (Unmanaged Code) امکان غیر فعال کردن این کنترل بر روی اندیس آرایه وجود دارد. برای مطالعه بیشتر در این زمینه میتوانید به فصل چهاردهم از کتاب "*Applied Microsoft .Net Framework Programming*" نوشته Jeffrey Richter مراجعه نمایید.

بهنگام کار با آرایه ها ممکن است نیاز به افزایش تعداد عناصر آرایه داشته باشیم. برای انجام چنین کاری لازم است تا نمونه جدیدی از آرایه مورد نظر را با طول جدید ایجاد کرده و عناصر آرایه فعلی را در آرایه جدید کپی نمایید. نمونه ای از کد مورد نظر برای اجرای چنین عملی در زیر آمده است :


```
// Create an integer array with three elements
int [] fib = new int[3];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;
// Redimension message to a 10 element array
int [] temp = new int[10];
// Copy the fib array to temp
fib.CopyTo(temp, 0);
// Assign temp to fib
fib = temp;
```

پس از اجرای آخرین خط در کد بالا، آریه fib به یک آریه ده عنصری اشاره میکند. عناصر 3 تا 9 این آریه جدید مقدار پیش فرض Int32 یعنی صفر را دارا خواهند بود.

آرایه ها یکی از بهترین ساختمان های داده برای نگهداری مجموعه ای از داده های همگون است که باید بطور مستقیم مورد دسترسی قرار گیرند. جستجو درون یک آرایه نا مرتب، زمان اجرای خطی دارد. توجه نمایید، در صورتیکه آرایه ای مرتب شده را مورد جستجو قرار دهیم، زمان اجرای جستجو به $O(\log n)$ تغییر پیدا خواهد کرد که زمانی نزدیک به زمان اجرایی درخت جستجوی دودویی است. توجه نمایید که در Net، کلاس Array دارای متد BinarySearch() است که با استفاده از آن میتوان به جستجوی باینری درون آرایه مرتب شده پرداخت.

نکته : همانطور که میدانید Net Framework. از آرایه های چند بعدی نیز پشتیبانی مینماید. از آنجائیکه زمان اجرای جستجو درون آرایه تک بعدی برابر با $O(n)$ است، زمان اجرای جستجو در یک آرایه k بعدی برابر با $O(n^k)$ خواهد بود. توجه کنید که در اینجا n تعداد عناصر آرایه در هر بعد است و منظور از n تعداد کل عناصر موجود در آرایه نیست.

ساخت ساختمان داده ای مطمئن، کارآ و قابل استفاده مجدد

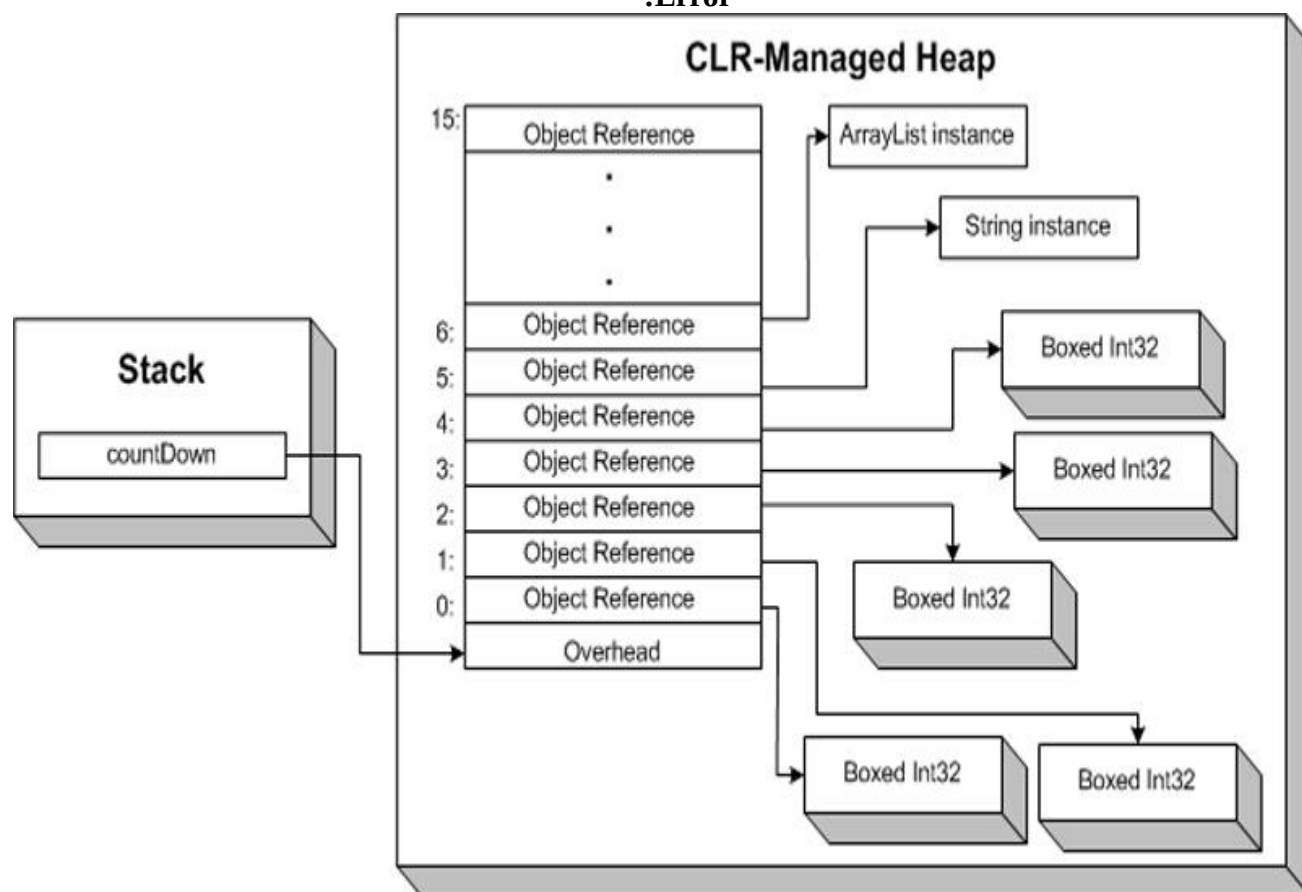
هنگامیکه میخواهیم ساختمان داده ای را برای مسئله ای خاص ایجاد نمائیم، ویژگیهای داخلی ساختمان داده در اغلب موارد، امکان شخصی سازی برای منطبق شدن با مسئله ما را فراهم مینماید. برای مثال فرض کنید میخواهید برنامه ای برای محاسبه حقوق کارکنان خود ایجاد نمایید. در این برنامه یکی از ساختمان داده های مورد نیاز میتواند آرایه Employee باشد. در عین حال شما نمی خواهید برای ورود هر کارمند جدید به شرکت، طول آرایه را تغییر دهید. از اینرو می توانید ساختمان داده ای شخصی تولید کنید که بصورت داخلی از یک آرایه استفاده می کند و کارآیی آن را افزایش میدهد. برای مثال میتوان به تواناییهای آرایه، قابلیت افزایش حجم و جستجو را اضافه کرد.

ایجاد چنین ساختمان داده ای آنچنان می تواند مفید باشد که ممکن بخواهید از آن در برنامه های دیگر خود نیز استفاده نمایید. اما این ساختمان داده برای استفاده در برنامه های دیگر مفید نیست چراکه تنها عناصری از نوع Employee را میپذیرد. یک روش برای انعطاف پذیر کردن ساختمان داده مورد نظر آنست که بجای استفاده از آرایه ای از نوع Employee، آرایه ای از نوع Object را درون ساختمان داده خود استفاده کنیم. چون تمام انواع در .Net Framework از نوع Object مشتق میشوند، از اینرو آرایه ما میتواند هر نوع خاصی را بپذیرد. با این روش میتوانیم از ساختمان داده ایجاد شده در برنامه های دیگر نیز استفاده کنیم.

در .Net Framework چنین ساختمان داده ای بطور از پیش تعریف شده وجود دارد: کلاس `ArrayList` در `System.Collections`. `ArrayList` درون خود از آرایه ای از نوع Object استفاده کرده و امکان تغییر سائز خودکار را با افزایش تعداد عناصر مورد نیاز، فراهم مینماید. به دلیل اینکه `ArrayList` از آرایه ای از نوع Object استفاده می کند، از اینرو برنامه نویسان میتوانند از آن برای کار با انواع مختلف داده ای، بهره گیرند.

ایجاد انعطاف پذیری بیشتر، هزینه هایی دارد و در مورد `ArrayList` این هزینه مربوط به کارایی است. از آنجائیکه `ArrayList` از نوع Object استفاده میکند، بهنگام خواندن مقادیر از `ArrayList` میبایست بطور صریح عمل casting را با توجه به نوع داده ذخیره شده انجام داد. (Casting عملی است که در آن نوع داده مورد استفاده را مشخص میکنیم. برای مثال `x = (int) y` در اینجا معین کرده ایم که متغیر `y` با نوع `int` به متغیر `x` تخصیص داده شود.) همانطور که به یاد دارید، گفتیم که آرایه هایی از نوع مقداری (همانند `System.Int32`، `System.Double` و `System.Boolean`) بصورت پیوسته در `Managed Heap` در فرم `Unboxed` خود ذخیره میشوند. اما آرایه مورد استفاده در `ArrayList` از نوع Object است. از اینرو هنگامیکه `ArrayList` مورد نظر از انواع مقداری استفاده نماید، هر عنصر `ArrayList` مرجعی به نوع مقداری در فرم `Boxed` خود خواهد بود. این مطلب در شکل زیر به نمایش گذاشته شده است.

!Error



Boxing و Unboxing ، هنگامیکه از ArrayList هایی با انواع مقداری استفاده می کنید و حجم دسترسی به عناصر ArrayList بالاست، میتوان بشکل محسوسی در کارایی برنامه تاثیر گذارد. همانطور که در شکل فوق نیز مشاهده میکنید، برای انواع مرجعی، فرمت شکل گیری حافظه در آرایه و ArrayList یکسان است.

استفاده از ArrayList ها، همچنین میتواند باعث بروز خطاهایی شود که تا زمان اجرای برنامه قابل لمس نخواهند بود. از آنجائیکه آرایه داخلی ArrayList از نوع Object است، قرار دادن نوع ناخواسته در ArrayList از سوی برنامه قابل قبول بوده و این مسئله باعث ایجاد خطا در برنامه میشود. همچنین این مسئله میتواند باعث پنهان شدن خطا تا زمان اجرا، تست و حتی، در بدترین شرایط، زمان تحویل نهایی برنامه شود.

Generic ها به کمک می آیند !

مسئله نوع و کارایی در ArrayList ها، با استفاده از Generic ها در Net Framework 2.0. بهبود یافته است. Generic این امکان را به طراح نرم افزار میدهد تا بهنگام ایجاد ساختمان داده انواع خاصی را برای استفاده در آن انتخاب نماید. برای روشن شدن بهتر موضوع مثالی را بررسی میکنیم. در این مثال کلاسی ایجاد میشود که در آن آرایه ای که به نوع خاصی محدود شده است، مورد استفاده قرار گرفته است.

```
public class TypeSafeList<T>
{
```

```

T[] innerArray = new T[0];
int currentSize = 0;
int capacity = 0;
public void Add(T item)
{
    // see if array needs to be resized
    if (currentSize == capacity)
    {
        // resize array
        capacity = capacity == 0 ? 4 : capacity * 2; // double capacity
        T[] copy = new T[capacity]; // create newly sized array
        Array.Copy(innerArray, copy, currentSize); // copy over the array
        innerArray = copy; // assign innerArray to the new, larger array
    }
    innerArray[currentSize] = item;
    currentSize++;
}
public T this[int index]
{
    get
    {
        if (index < 0 || index >= currentSize)
            throw new IndexOutOfRangeException();
        return innerArray[index];
    }
    set
    {
        if (index < 0 || index >= currentSize)
            throw new IndexOutOfRangeException();
        innerArray[index] = value;
    }
}
public override string ToString()
{
    string output = string.Empty;
    for (int i = 0; i < currentSize - 1; i++)
        output += innerArray[i] + ", ";
    return output + innerArray[currentSize - 1];
}
}

```

در نخستین خط این کد و درون کلاس `TypeSafeList`، شناسه ای جهت تعیین نوع `T` تعریف شده است. این کد معین میکند که استفاده کننده از آن میبایست یک نوع خاص را برای کار با آن در نظر بگیرد. در این کد، این نوع قابل انتخاب توسط برنامه نویس، با عنوان `T` تعریف شده که میتوان به جای `T` از هر نام متغیر دیگری نیز استفاده کرد. از این

متغیر در متدها و ویژگیهای کلاس نیز استفاده شده است. برای مثال، آرایه داخلی از نوع T تعریف شده و متد Add متغیری از نوع T را بعنوان ورودی میپذیرد.

برای تعریف متغیری از این کلاس، برنامه نویس میبایست نوع T را معین نماید :

TypeSafeList<type> variableName;

در قطعه کد زیر نیز، نمونه ای از کلاس TypeSafeList که نوع int را میپذیرد نشان داده شده است :

```
TypeSafeList<int> fib = new TypeSafeList<int>();  
fib.Add(1);  
fib.Add(1);  
for (int i = 2; i < 25; i++)  
fib.Add(fib[i - 2] + fib[i - 1]);  
Console.WriteLine(fib.ToString());
```

از مهمترین مزایای Generics به شرح زیر است :

Type-Safe : با استفاده از کلاس SafeTypeList، برنامه نویس اطمینان حاصل میکند که از انواع ناخواسته به طور اشتباه استفاده نخواهد کرد.

کارآیی : کنترل نوع مورد استفاده در زمان اجرا را حذف کرده و هزینه مربوط به Boxing و Unboxing را کاهش میدهد.

قابلیت استفاده مجدد : قابلیت استفاده مجدد در برنامه های مختلف را بالا میبرد.

بسیاری از ساختمان داده هایی که در این سری مطالب مورد بررسی قرار خواهند گرفت از Generics ها استفاده میکنند و بهنگام ساخت یک ساختمان داده، مانند درخت دودویی که در قسمت سوم مورد بررسی قرار خواهد گرفت، از Generics استفاده خواهیم کرد.

لیست : آرایه ای همگون با قابلیت تغییر سائز ابعاد

یک آرایه، همانطور که قبلا نیز مشاهده کردیم، ساختمان داده ای با قابلیت نگهداری تعداد معینی عنصر هم نوع است که بطور مرتب در کنار یکدیگر قرار گرفته اند. هر چند استفاده از آرایه بسیار ساده است، اما در مواردی همچون زمانیکه تعداد عناصر آرایه برای ما مشخص نیست و یا نیاز به تغییر سائز آرایه وجود داشته باشد، میواند مشکل ساز باشد. یک روش برای پرهیز از تغییر سائز آرایه بطور دستی، ایجاد ساختمان داده ای مشابه به آرایه است که علاوه بر امکان خواندن و نوشتن اعضا، قابلیت افزایش سایر را نیز در موارد مورد نیاز فراهم نماید. راهکار Net Framework برای این ساختمان داده، لیست است که در Syssem.Collections.Generics قرار دارد.

لیست، کلاسی است که درون خود از یک آرایه بهره می برد و علاوه بر قابلیت خواندن و نوشتن اعضاء این آرایه داخلی، قابلیت تغییر سائز خودکار آرایه را نیز فراهم می نماید. لیست نیز همانند آرایه، ساختمان داده ای همگون است، بدین معنا که تنها قابلیت ذخیره سازی عناصر هم نوع (از یک نوع داده ای) را دارا می باشد. لیست، از Generics بهره میبرد تا با استفاده از آنها، برنامه نویس امکان تعیین نوع داده مورد نظر را در هنگام استفاده، دارا باشد. در هنگام ایجاد نمونه ای جدید از کلاس لیست، میبایست نوع داده مورد استفاده در آن تعیین گردد :

// int ساخت لیستی از نوع داده ای

List<int> myFavoriteIntegers = new List<int> ();

// ساخت لیستی از نوع داده ای رشته

List<string> friendsNames = new List<string> ();

توجه کنید که نوع داده مورد استفاده در لیست بهنگام ایجاد نمونه ای جدید از کلاس لیست مشخص می گردد. بهنگام ایجاد نمونه جدید از لیست نیازی به تعیین سائز لیست نیست، هر چند میتوان سائز پیش فرضی برای لیست در نظر گرفت. برای افزودن عنصری به لیست، تنها کافیت تا از متد Add() آن استفاده کنید. همانند آرایه، دسترسی به عناصر لیست نیز از طریق اندیس صورت می گیرد. در مثال زیر لیستی از int ایجاد شده است. سپس با استفاده از متد Add() عناصری به آن افزوده شده و سپس با استفاده از اندیس به این عناصر دسترسی پیدا شده است .

// ساخت لیست

List<int> powersOf2 = new List<int> ();

// افزودن 6 عنصر به لیست

powersOf2.Add(1);

powersOf2.Add(2);

powersOf2.Add(4);

powersOf2.Add(8);

powersOf2.Add(16);

powersOf2.Add(32);

// تغییر دومین عنصر لیست به مقدار 10

powersOf2[1] = 10;

// انجام یک عملیات ریاضی با استفاده از عناصر لیست

int sum = powersOf2[2] + powersOf2[3];

لیست، در حقیقت پیدا سازی خاص از یک آرایه است که پیچیدگی های پیاده سازی را پنهان کرده است. با استفاده از لیست، نیازی به تعیین سائز آرایه در ابتدا ایجاد آن نداریم. همچنین به هنگام افزودن عنصری به آن، نیازی به تغییر سائز آرایه داخلی آن نیست. علاوه بر مطالب گفته شده، لیست متدهایی جهت انجام عملیات مورد استفاده بر روی آرایه ها می باشد. برای مثال، در یک آرایه، برای پیدا کردن یک عنصر درون اعضای آرایه می بایست از حلقه for استفاده نمایید. در

لیست با استفاده از متد `Contains()` می توان به وجود عنصری خاص در لیست پی برد. همچنین با استفاده از متد `IndexOf()` می توان به محل ذخیره سازی عنصری خاص در لیست پی برد. کلاس لیست همچنین دارای متد `BinarySearch()` است که با استفاده از آن می توان به جستجوی باینری درون لیست مرتب شده پرداخت. با استفاده از متدهای دیگری نظیر `Find()`، `FindAll()`، `Sort()` و `ConvertAll()` نیز می توان با استفاده از `delegate` ها، به اجرای عملیاتی پرداخت که اجرای آنها بر روی آرایه ها نیاز به نوشتن چندین خط کد دارد.

زمان اجرای لیست، همانند زمان اجرایی یک آرایه استاندارد است، هر چند لیست توان پردازشی بیشتری نسبت به آرایه نیاز دارد.

نتیجه گیری :

در این مطلب، به عنوان اولین مطلب از سری مطالب مربوط به ساختمان های داده، با مفهوم ساختمان داده و اهمیت استفاده از آنها و همچنین با نحوه آنالیز کارایی یک ساختمان داده و اهمیت آن، آشنا شدیم.

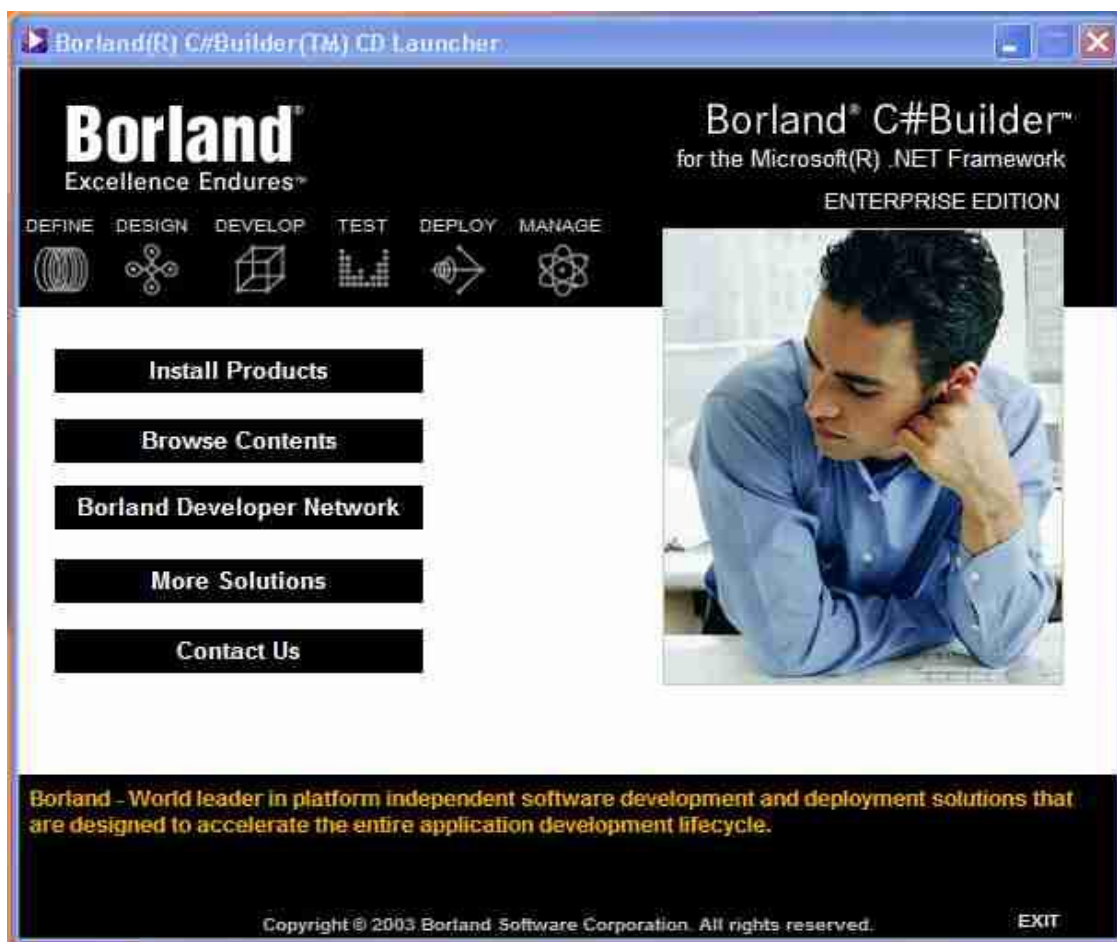
پس از آشنایی با چگونگی آنالیز کارایی ساختمان های داده، به بررسی دو ساختمان داده مهم و پر کاربرد در `.Net` یعنی `Framework Base Class Library` و `System.Collections.Generic.List` پرداختیم. آرایه ساختمان داده ای همگون بود که با استفاده از آن به سادگی می توانیم به عناصر آن دسترسی پیدا کرده و عمل خواندن و نوشتن را انجام دهیم. نقطه ضعف آرایه در جستجوی عنصری خاص در آن بود. همچنین افزایش سایز آرایه نیز نیازمند نوشتن چندین خط کد بود.

کلاس لیست، با استفاده از چند متد، کارایی آرایه را افزایش داد. برای مثال متد `Add()` عنصری را به لیست اضافه می کند و سایز لیست را بطور خودکار افزایش می دهد. همچنین با استفاده از متد `IndexOf()` می توان اندیس عنصری خاص در لیست را بدست آورد. زمان اجرایی لیست چندان تفاوتی با آرایه ندارد اما کار برنامه نویس را در پیدا سازی بسیار آسان می نماید.

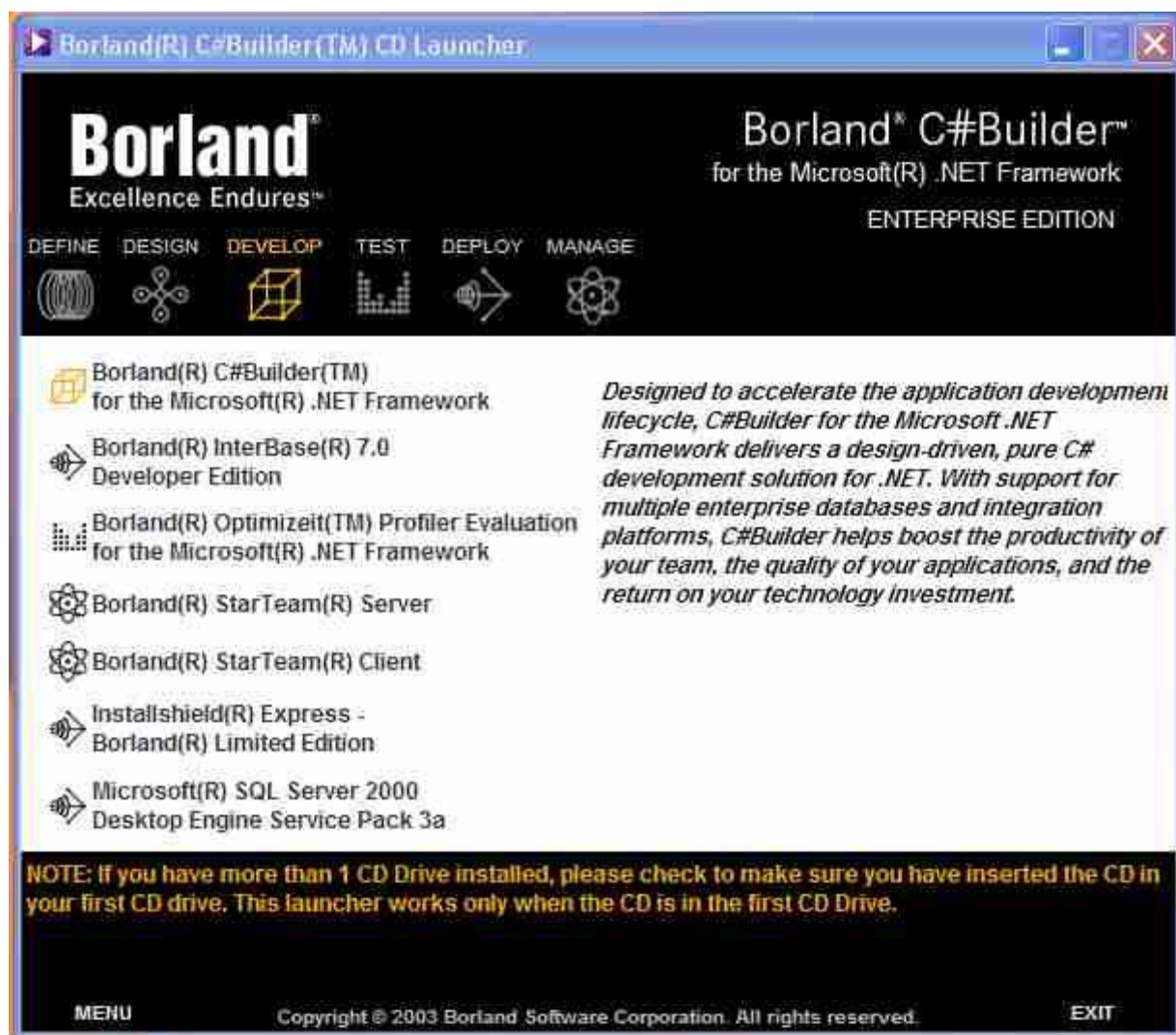
در قسمت بعدی با کلاس پشته (`Stack`) و صف (`Queue`) آشنا می شویم. در قسمت بعد همچنین با آرایه های شرکت پذیر آشنا می شویم. این آرایه ها از کلید هایی رشته ای بعنوان اندیس استفاده می کنند و در `.Net Framework` با استفاده از کلاسهای `Dictionary` و `Hashtable` ایجاد شده اند.

نصب C#Builder

در این قسمت اطلاعاتی راجع به نصب برنامه در اختیار شما قرار خواهد گرفت. نسخه Enterprise نرم افزار C#Builder، بر روی دو CD ارائه شده است. CD اول مربوط به خود برنامه و CD دوم برنامه‌های و محصولات جانبی مرتبط با C#Builder است. پس از قرار دادن CD شماره 1 درون CD-ROM، صفحه‌ای همانند شکل زیر ظاهر می‌گردد.



پس از باز شدن این صفحه بر روی گزینه **Install Products** کلیک کنید تا صفحه‌ای همانند شکل زیر نمایان گردد. (نکته: توجه نمایید که در صورتیکه دارای دو CD-ROM بر روی سیستم خود هستید، برای اجرای برنامه و نصب آن حتماً از اولین CD-ROM موجود بر روی سیستم خود استفاده نمایید در غیر اینصورت نمی‌توانید برنامه را نصب نمایید و پس از کلیک بر روی گزینه‌های نصب هیچ اتفاقی رخ نداده و پیغام خطایی ظاهر می‌شود.)



حال با انتخاب اولین گزینه "Borland C#Builder for Microsoft .Net Framework" نصب برنامه آغاز می‌شود. سایر گزینه‌ها مربوط به برنامه‌ها و ابزارهای جانبی است که به همراه C#Builder عرضه شده‌اند و پس از نصب برنامه می‌توانید آنها را نیز نصب کنید.

نکته : در صورتیکه پس از کلیک کردن بر روی گزینه اول جهت نصب برنامه با پیغام خطایی بشکل :

"The 'Prereqs.exe' application could not be located. Please insert the other disk and press OK to continue" مواجه شدید، CD برنامه را در درایو دیگر خود قرار دهید.

قبل از نصب C#Builder لازم است تا موارد زیر نیز بر روی سیستم شما نصب شده باشند :

1- .Net Framework

2- .Net Framework SDK

Internet Explorer 6 with SP1 - 3

ASP.Net - 4 جهت اجرای برنامه‌های

Microsoft Visual J#.Net - 5

در صورتیکه این برنامه‌ها بر روی سیستم شما وجود نداشته باشند، برخی از آنها را می‌توانید از روی CD نصب برنامه نصب نمایید. بعنوان مثال **.Net Framework** و **.Net Framework SDK** و نیز **J#.Net** بطور خودکار نصب می‌شوند. اما در صورتیکه **IIS** بر روی سیستم شما نصب نیست باید با استفاده از CD نصب ویندوز XP (در صورتیکه از ویندوز XP استفاده می‌کنید) آنرا بر روی سیستم خود نصب نمایید. برای نصب **IIS** وارد **Control Panel** شوید و گزینه **Add/Remove Windows Components** را انتخاب نمایید. سپس صفحه‌ای باز می‌شود که در آنجا می‌توانید اجزای مختلفی را به ویندوز خود بیفزایید. (البته نصب **IIS** تنها در صورتی ضروری است که مایل به برنامه نویسی تحت **ASP.Net** باشید).

اگر **.Net Framework** و **SDK** را بر روی سیستم خود ندارید، در ابتدای نصب از شما سوالی مبنی بر نصب این دو پرسیده می‌شود، با انتخاب گزینه **Yes** این دو برای شما نصب خواهند شد.

پس از اتمام نصب **.Net Framework** و **J#.Net** نوبت به نصب **C#Builder** می‌رسد. برای نصب برنامه سه گزینه در پی رو خواهید داشت: **Typical**, **Minimum** و **Custom**. در صورتیکه گزینه **Typical** را انتخاب نمایید اکثر قسمت‌ها و اجزای برنامه بر روی سیستم شما نصب خواهند شد. انتخاب **Minimum** تنها آن قسمت از برنامه را نصب می‌نماید که برای اجرای محیط و تولید نرم‌افزار ضروری است و در نهایت با استفاده از گزینه **Custom** می‌توانید تک تک اجزای قابل نصب بر روی سیستم را مشاهده و سپس در صورت تمایل آنها را نصب نمایید. یک توصیه به دوستان اینست که در اگر با اجزای این محیط آشنایی چندانی ندارید گزینه **Typical** را انتخاب نمایید. انتخاب گزینه **Minimum** بدترین حالت نصب برنامه است چراکه بسیاری از امکانات این محیط نصب نخواهند شد. نظر شخصی من نصب برنامه با استفاده از گزینه **Custom** است. اگر برنامه را بصورت **Custom** نصب می‌کنند، توصیه می‌کنم تا تمامی امکانات موجود را برای نصب انتخاب نمایید. از جمله جالب ترین این امکانات **Code Visualization** است که بوسیله این مولفه قادر خواهید بود تا دیاگرام **UML** برنامه خود را در حین طراحی و ساخت نرم‌افزار مشاهده نمایید. پس از انتخاب این مولفه صفحه‌ای مطابق با شکل زیر ظاهر می‌گردد که مربوط به نرم‌افزارها و یا مولفه‌های جانبی هستند که به همراه **C#Builder** عرضه شده‌اند. البته توجه نمایید این مولفه‌ها بسته به نوع بسته نرم‌افزاری **C#Builder** متفاوت خواهند بود. در نسخه **Enterprise** بیشترین تعداد برنامه‌ها و مولفه‌های جانبی وجود دارند. (البته در ایران فقط نسخه **Enterprise** موجود است که در ضمن بهترین بسته نرم‌افزاری **C#Builder** است).



نکته دیگری که لازم است در اینجا به آن اشاره کنم در مورد نصب **Borland Janeva** است. اگر بر روی سیستم خود یکی از کامپایلرهای جاوا نظیر **Microsoft J++** و یا **Borland JBuilder** را دارید، توجه نمایید تا در هنگام نصب **Janeva** صفحه‌ای باز می‌شود و از شما درباره تنظیم کردن **JRE** سوالاتی می‌پرسد و شما باید از میان سه گزینه موجود یکی را انتخاب نمایید. اگر با تنظیمات **JRE** بطور کامل آشنایی ندارید گزینه **Don't configure JRE** را انتخاب نمایید زیرا در برخی موارد تنظیم **JRE** در این مرحله باعث ایجاد تغییراتی در محیط اجرایی کامپایلر جاوا شده و در هنگام اجرای کامپایلر جاوا با خطاهایی نظیر **JRE Failed to initialize** و یا **JITD ERROR** و نظایر آن مواجه خواهید شد. چون بازگرداندن سیستم با حالت قبلی بسیار دشوار بوده و نیاز به تخصص تکنیکی فراوان دارد توصیه می‌شود در صورت نداشتن چنین تخصصی از گزینه سوم جهت نصب استفاده کنید چون تنظیم **JRE** در این مرحله حتی باعث ایجاد **Restore Point** در ویندوز نیز نخواهد شد!!!!

در اینجا نصب برنامه به پایان می‌رسد و می‌توانید آنرا اجرا نمایید. اگر مایل به نصب سایر برنامه‌های جانبی باشید می‌توانید **CD** شماره یک برنامه را مجدداً راه‌اندازی کرده و در قسمت نصب، برنامه‌های موجود را برای نصب انتخاب نمایید.

شروع پروژه با **C#Builder**

هدف اصلی **C#Builder** بالا بردن بهره‌وری هنگام ایجاد کد است. به همین منظور، تمام کارهای متداولی را که اکثر توسعه‌دهندگان نرم‌افزار انجام می‌دهند را به صورت خودکار انجام می‌دهد. بعنوان مثال تمام پروژه‌ها با

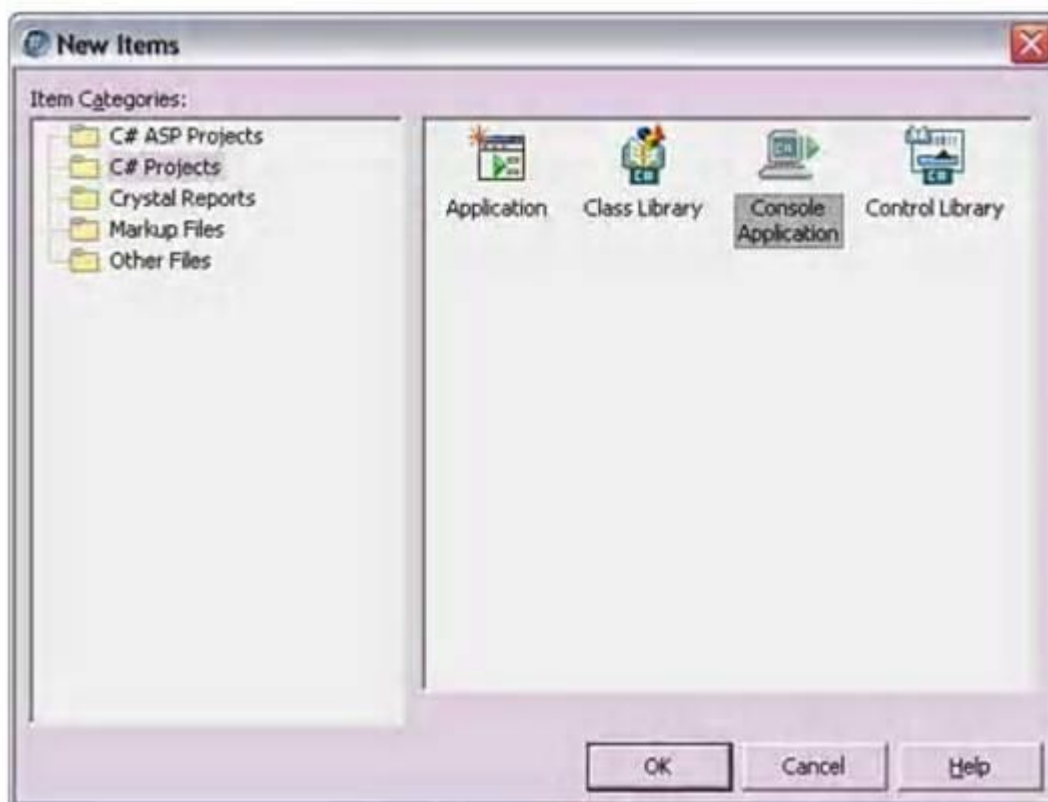
یکسری از کدهای مشخص و معین آغاز می‌شوند. در حالت عادی نویسنده برنامه باید این کدها را در هربار نوشتن برنامه، تایپ کرده و سپس به نوشتن سایر کدها بپردازد. **C#Builder** با هر بار آغاز پروژه‌ای جدید، تمامی کدهای لازم جهت آغاز به کار آن پروژه خاص را به طور خودکار تولید می‌نماید و دیگر برنامه‌نویس نیازی به تایپ مجدد آنها ندارد. ویزاردها از دیگر قابلیت‌های مهم این محیط هستند. با وجود ویزاردهای موجود در این محیط می‌توان به راحتی به ایجاد پروژه‌هایی جدید از قبیل: برنامه‌های کنسول، فرم‌های ویندوز، **ASP.Net** و **XML** پرداخت.

C#Builder امکان هر کاری را دارد!

سوالی که اغلب مردم درباره **C#Builder** می‌پرسند، اینست که چه کارهایی را می‌توانند توسط آن انجام دهند. افراد مبتدی تصور می‌کنند در صورتیکه برای نوع خاصی از برنامه ویزاردی موجود نباشد، دیگر قادر به ساخت آن نوع خاص برنامه نخواهند بود. باید گفت که اصلاً چنین نیست، چراکه هر نوع برنامه قابل پیاده‌سازی تحت **.Net**. بوسیله **C#Builder** قابل اجرا و پیاده‌سازی است. در صورتیکه برای کار خاص شما ویزاردی موجود نیست، می‌توانید با آغاز یک پروژه خالی (**Blank Project**) کدهای مورد نظر خود را پیاده‌سازی نمایید. شما همچنین می‌توانید خود به ساخت ویزارد بپردازید.

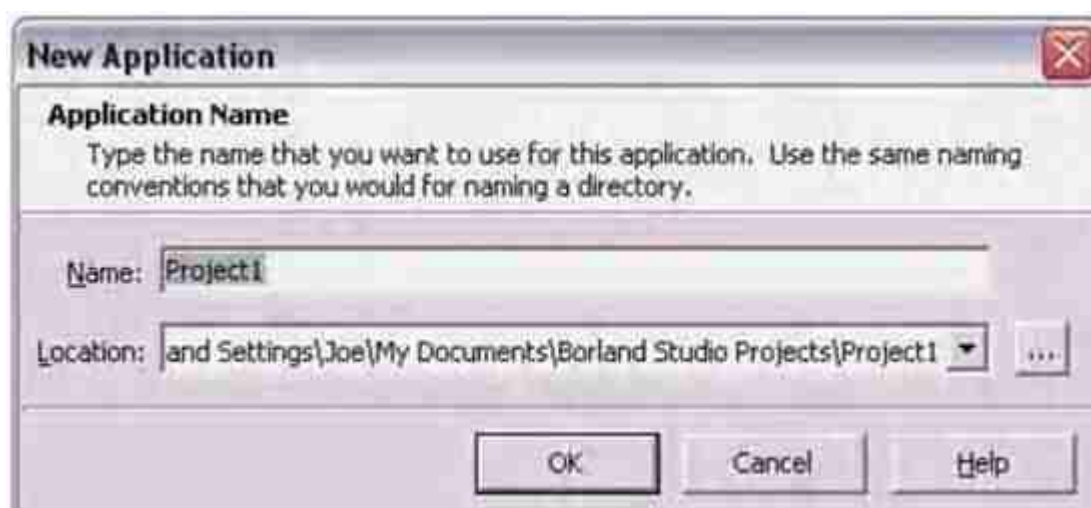
ساده‌ترین روش برای شروع یک پروژه جدید استفاده از لینک موجود در صفحه اول (به شکل 1-2 رجوع شود) است. برای شروع روی لینک **New** کلیک کنید. این کار دیالوگ ساخت پروژه جدید را باز می‌کند. همانطور که در شکل 1-3 نشان داده شده است، در این پنجره امکان انتخاب پروژه‌هایی از انواع مختلف وجود دارد. روش دیگر جهت شروع پروژه‌ای جدید انتخاب منوی **File** و سپس انتخاب گزینه **New** است.

داخل پنجره باز شده در شکل 1-3، انواع پروژه‌های از پیش تعیین شده **C#Builder** در سمت چپ دسته‌بندی شده‌اند. با کلیک بر روی هر یک از آنها ویزاردهای موجود ظاهر می‌شوند. برای شروع **Console Application** را انتخاب کنید و **OK** را کلیک نمایید.



شکل 1-3 پنجره انتخاب نوع پروژه جدید مورد نظر

با انتخاب این گزینه، پنجره دیگری همانند شکل 1-4 ظاهر می‌گردد که در آن نام و محل ذخیره‌سازی پروژه مشخص می‌گردد. بصورت پیش فرض، محل ذخیره‌سازی پروژه‌ها، دایرکتوری پروژه‌های بورلند در **My Documents** می‌باشد. شما می‌توانید هر محل دیگری را جهت ذخیره‌سازی پروژه‌های خود معین نمایید.



شکل 4-1 تعیین نام و محل ذخیره سازی پروژه

همانند سایر ویزاردها، گروه پروژه، پروژه و فایل های پیش فرض مورد نیاز به صورت خودکار ساخته می شوند. فایلی که برای یک برنامه کنسول ساخته می شود یک سورس کد **C#** است که معمولاً تحت نام **class.cs** ذخیره می گردد. پسوند فایل های ذخیره شده **C#** بصورت **.cs** است. **class.cs** شامل اسکلت کدهای لازم جهت شروع اجرای یک برنامه است. این فایل قابل کامپایل شدن و اجرا است اما هیچ کاری انجام نمی دهد. نام فایل نیز قابل تغییر است. بعنوان مثال در اینجا نام این فایل را از **class.cs** به **Welcome.cs** تغییر می دهیم. در هنگام تغییر نام فایلها، تنها نکته مهم اینست که پسوند فایل تغییر نکند زیرا در این صورت فایل از سوی **C#Builder** قابل شناسایی نخواهد بود.

برای اجرای برنامه، منوی **Run** و سپس **Run Without Debugging** را انتخاب نمایید. پس از این مرحله برنامه کامپایل شده و برنامه به سرعت اجرا شده و صفحه ای به سرعت چشمک زده و بسته می شود. در بخش بعدی نوشتن یک برنامه ساده را توضیح می دهیم.

اجرای برنامه ها

C#Builder امکانات متفاوتی جهت ساخت و اجرای برنامه ها دارا می باشد. گزینه هایی که در اختیار برنامه نویس قرار می گیرند، بستگی مسقیم با کاری دارند که می خواهید انجام دهید.

یک برنامه نمونه

برای شروع، یک برنامه بسیار ساده، که داری حداقل اجزای برنامه است را در مثال زیر نشان داده ایم. این مثال یک برنامه بسیار ساده **C#** است.

```
using System;
```

```
namespace CSharp-Persian
```

```
{
```

```
    class Welcome
```

```
    {
```

```

static void Main()
{
    Console.WriteLine("Welcome to C#-Persian");
    Console.ReadLine();
}
}

```

این برنامه با استفاده از کلمه کلیدی **using** آغاز شده که در آن دسترسی به کلاسهای مختلف **System** را امکان پذیر می کند. **System** شامل نوعها و کلاسهای پایه ای است که در اغلب برنامه های **C#** مورد استفاده هستند. کل این برنامه تحت نام **CSharp-Persian** شناخته می شود. با استفاده از کلمه کلیدی **namespace** نام برنامه مشخص شده است. دو آکولاد باز و بسته **{}**، آغاز و خاتمه برنامه یا یک بلوک از کد را نشان می دهند.

تمامی برنامه های **C#**، دارای حداقل یک کلاس اصلی هستند که متد **Main()** را در بر می گیرد. در این برنامه، این کلاس **Welcome** نام دارد. متد **Main()** نقطه آغازین این برنامه (و هر برنامه دیگر **C#**) است. اسمبلی هایی که دارای متد **Main()** هستند در حالت عادی، فایل هایی اجرایی (**Executable**) هستند و اسمبلی هایی که فاقد این متد باشند از نوع کتابخانه ای (**dll**) می باشند. پراوتزهای جلوی این متد خالی هستند که بیانگر این مطلب است که این متد هیچ پارامتری دریافت نمی کند (در این مثال). نوع مقدار بازگشتی این متد از نوع **void** تعریف شده است که بیانگر آنست که متد **Main()** هیچ مقداری بر نمی گرداند.

در **C#**، دو روش برای فراخوانی کد موجود است، بوسیله نوع (**Type**) و بوسیله نمونه (**Instance**). اولین باری که یک نوع مورد استفاده قرار می گیرد، **CLR** آنرا در حافظه بارگذاری می کند. اگر این نوع در دسترس باشد، در حافظه می ماند و کد اجرا می گردد. اگر نوع بصورت نمونه ای باشد، نمونه ای از آن نوع در حافظه بارگذاری می شود. در این حالت تنها آن اعضای از نوع که بصورت **static** تعریف شده اند از طریق نوع قابل دسترسی می باشند.

علت مطرح کردن این مطلب در اینجا آنست که متد **Main()** باید به صورت **static** تعریف گردد. به دلیل اینکه متد **Main()** اولین کدی است که در برنامه فراخوانی می شود و به همین دلیل به هیچ طریقی امکان ساخت نمونه ای

(instance) از آن قبل از اجرای برنامه وجود ندارد. پس این متد همیشه باید به صورت **static** تعریف شود. از

اینرو CLR متد **Main()** را با نوع (by type) بارگذاری می‌کند.

کدهای درون این متد هسته اصلی این برنامه هستند. این برنامه از کلاس **Console** استفاده می‌کند که این کلاس از

عملیات **I/O** در برنامه‌نویسی کنسول پشتیبانی می‌نماید. اکثر برنامه‌هایی که در بخشهای ابتدایی این سایت آورده

شده‌اند، به دلیل سادگی و فهم بهتر زبان، از کنسول **I/O** استفاده می‌کنند. در قسمتهای بعدی سایت از فرمهای ویندوز یا

فرمهای وب **ASP.Net** نیز استفاده خواهیم کرد. کنسول شامل 4 دستور ابتدایی **WriteLine**، **Write** جهت نوشتن

متن در خروجی و **Read** و **ReadLine** جهت خواندن از ورودی، می‌باشد. جدول 1-2 لیستی از اعضای ابتدایی

کلاس کنسول و کاربرد آنها را نشان می‌دهد. برای آگاهی از سایر اعضای این کلاس می‌توانید به راهنمای **C#Builder**

و یا مستندات **.Net Framework** مراجعه نمایید.

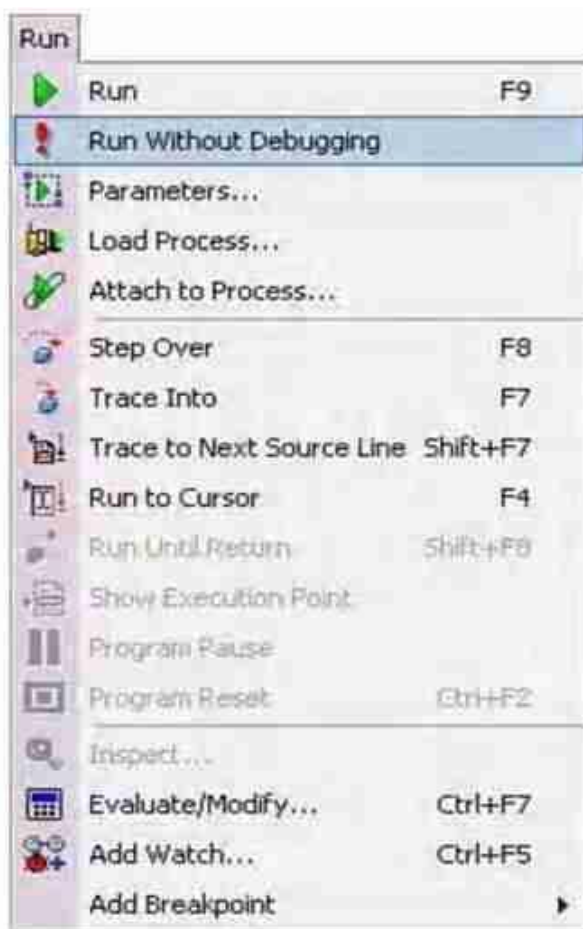
نام عضو	توضیحات
WriteLine	خروجی فرمت دار را به همراه یک خط جدید ایجاد می‌نماید. (برای نمایش داده‌ها در خروجی - مکان نما به خط بعدی می‌رود)
Write	خروجی فرمت دار را بدون ایجاد یک خط جدید تولید می‌نماید. (برای نمایش داده‌ها در خروجی - مکان نما در انتهای خروجی نمایش داده شده باقی می‌ماند)
ReadLine	رشته‌ای از داده‌های را تا هنگام فشردن کلید Enter از ورودی دریافت می‌دارد.
Read	داده‌های ورودی را کاراکتر به کاراکتر دریافت می‌دارد. فشردن کلید Enter در این حالت برای جداسازی کاراکترهای ورودی از یکدیگر مورد استفاده قرار می‌گیرد.

جدول 1-2 اعضای ابتدایی کلاس کنسول

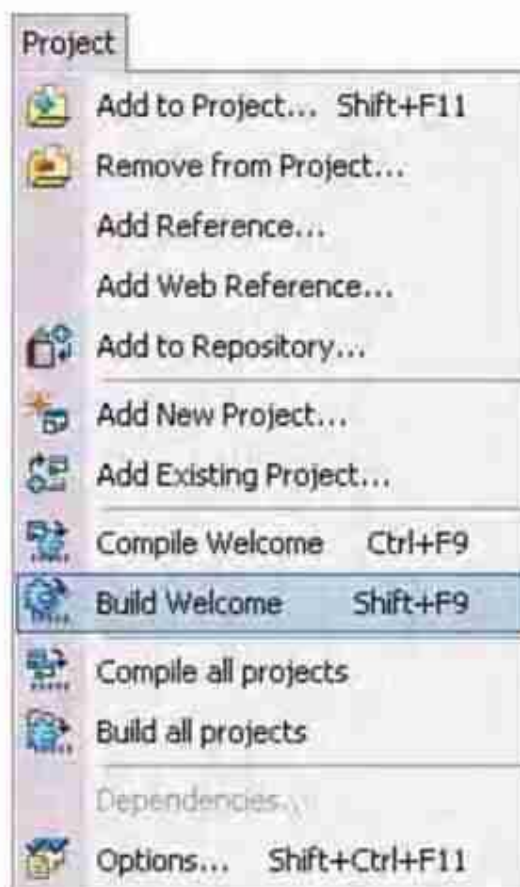
کامپایل و اجرا

با فشردن کلید **F9** برنامه کامپایل و اجرا می‌شود. پس از اجرای این برنامه عبارت **Welcome to C# Persian** بر روی صفحه نمایش ظاهر می‌شود. این عمل بوسیله دستور **Console.WriteLine** صورت می‌پذیرد. دستور بعدی در این برنامه **Console.ReadLine** است که این دستور منتظر دریافت عبارتی از ورودی می‌ماند و تا زمانی که کاربر عبارتی را وارد نکند، برنامه در همان نقطه منتظر می‌ماند. در صورتیکه از این دستور در انتهای برنامه استفاده نکنیم، برنامه به سرعت اجرا می‌شود و قادر به دیدن خروجی نخواهیم بود، چراکه خروجی برنامه به صورت آنی نمایش داده می‌شود و اجرای برنامه خاتمه می‌یابد.

گزینه دیگری که برای اجرای کد برنامه وجود دارد، گزینه **Run Without Debugging** موجود در منوی **Run** است. (شکل 1-5 را مشاهده نمایید). منوی **Project** نیز حاوی گزینه‌هایی جهت کامپایل و ساخت پروژه است. (به شکل 1-6 توجه کنید)



شکل 1-5 منوی **Run**



شکل 6-1 منوی Project

C#Builder چیست؟

C#Builder محیط یکپارچه توسعه یافته (IDE) جهت تولید و توسعه نرم افزار به زبان برنامه نویسی **C#** است که تحت محیط **.Net**. شرکت مایکروسافت کار می کنند. این محیط همچنین شامل ابزارهای متعددی است که امر برنامه نویسی را تسهیل می بخشد. در ادامه توضیح مختصری درباره این ابزارها داده می شود.

ساخت واسطه های کاربر (User Interfaces)

ابزارهای طراحی و ویژوال **C#Builder** این امکان را برای شما فراهم می کنند تا تنها با انتخاب مولفه های مختلف از پنجره **Tool Palette** و قرار دادن آنها بر روی فرم، به راحتی بتوانید واسطه های کاربر گرافیکی طراحی و ایجاد نمایید. با این ابزار می توانید فرمهای ویندوز، صفحات وب در فرمت **ASP.Net** و صفحات **HTML** ایجاد نمایید.

تولید و ویرایش کد

محیط ویرایش کد در **C#Builder** حاوی ابزارهایی است در امر تولید و نوشتن کدهای برنامه ها بسیار کارآمد و مفید می باشند. از نمونه های این ابزارها می توان به قطعه کدهای رایج و آماده در **Code Snippets** اشاره کرد که به راحتی می توان آنها در برنامه وارد نمود تا در وقت صرفه جویی گردد. همچنین این محیط دارای قابلیت می باشد که کاربر می تواند کدهای مورد نظر خود را که در طول برنامه به دفعات از آن استفاده می کند را در این قسمت وارد کرده و در

مواقع مورد نظر از آنها استفاده نماید. همچنین با استفاده از ماکروها می‌توانید اعمال مورد نظر خود را ضبط کرده و سپس در جایی دیگر از آنها استفاده نمایید. (درباره ماکروها توضیح داده خواهد شد).

کامپایلر، عیب‌یابی و توسعه نرم‌افزارها

درون IDE، می‌توانید برنامه‌های خود را کامپایل و اجرا نمایید. همچنین می‌توانید پیغام‌های اخطار، خطا و یا راهنمایی‌های کامپایلر را نیز مشاهده کنید. عیب‌یاب پیشرفته موجود در **C#Builder** خطایابی و رفع نقایص برنامه را آسان‌تر می‌کند و همچنین به شما در عیب‌یابی نقایص منطقی برنامه نیز کمک می‌نماید. همچنین در زمان اجرای برنامه می‌توانید به منظور مشاهده وضعیت متغیرها اجرای برنامه را متوقف سازید تا از این طریق روند اجرایی برنامه را نیز کنترل نمایید. **.Net Framework** حاوی ابزارهای متنوعی جهت توسعه و ارائه نرم‌افزارها می‌باشد. به همراه **C#Builder** نسخه‌ای از نرم‌افزار **InstallShield Express** ارائه شده که بوسیله می‌توانید برنامه خود را جهت نصب در ویندوز آماده نمایید.

کنترل دسترسی و پیگیری تغییرات اعمال شده در کد

سیستم کنترل سورس کد، این امکان را برای محیط‌های کاری تیمی فراهم می‌آورد تا از نحوه دسترسی و اعمال تغییرات در کد برنامه‌ها مطلع گردید. **C#Builder** از **API** کنترل سورس کد مایکروسافت (**SCC API**) جهت ایجاد واسطی به **StarTeam**، **CVS**، **ClearCase** و **Visual SourceSafe**، استفاده می‌نماید. درون خود IDE نیز می‌توانید کارهای مربوط به کنترل سورس کد را انجام دهید.

.Net Framework

.Net Framework. شرکت مایکروسافت تکنولوژی پایه و ساختار زیر بنایی تولید و اجرای برنامه‌های **.Net** است. **Framework** حاوی **(Common Language Runtime) CLR** و کتابخانه کلاسها (**Class Library**) است. **CLR** مدیریت اجرای برنامه‌ها را بر عهده دارد و سرویس‌هایی از قبیل مدیریت حافظه را بر عهده می‌گیرد. کتابخانه کلاسها مجموعه‌ای از مولفه‌های (**Componets**) شی‌گرا قابل استفاده مجدد برای تولید و توسعه برنامه‌های **.Net** می‌باشد.

C#Builder تمامی کتابخانه کلاسهای **Framework** را از طریق IDE در دسترس قرار می‌دهد تا بتوانید برنامه‌ها را سریع‌تر و بهتر تولید و ایجاد نمایید. **C#Builder** در موارد زیر به بسط و توسعه **Framework** پرداخته است:

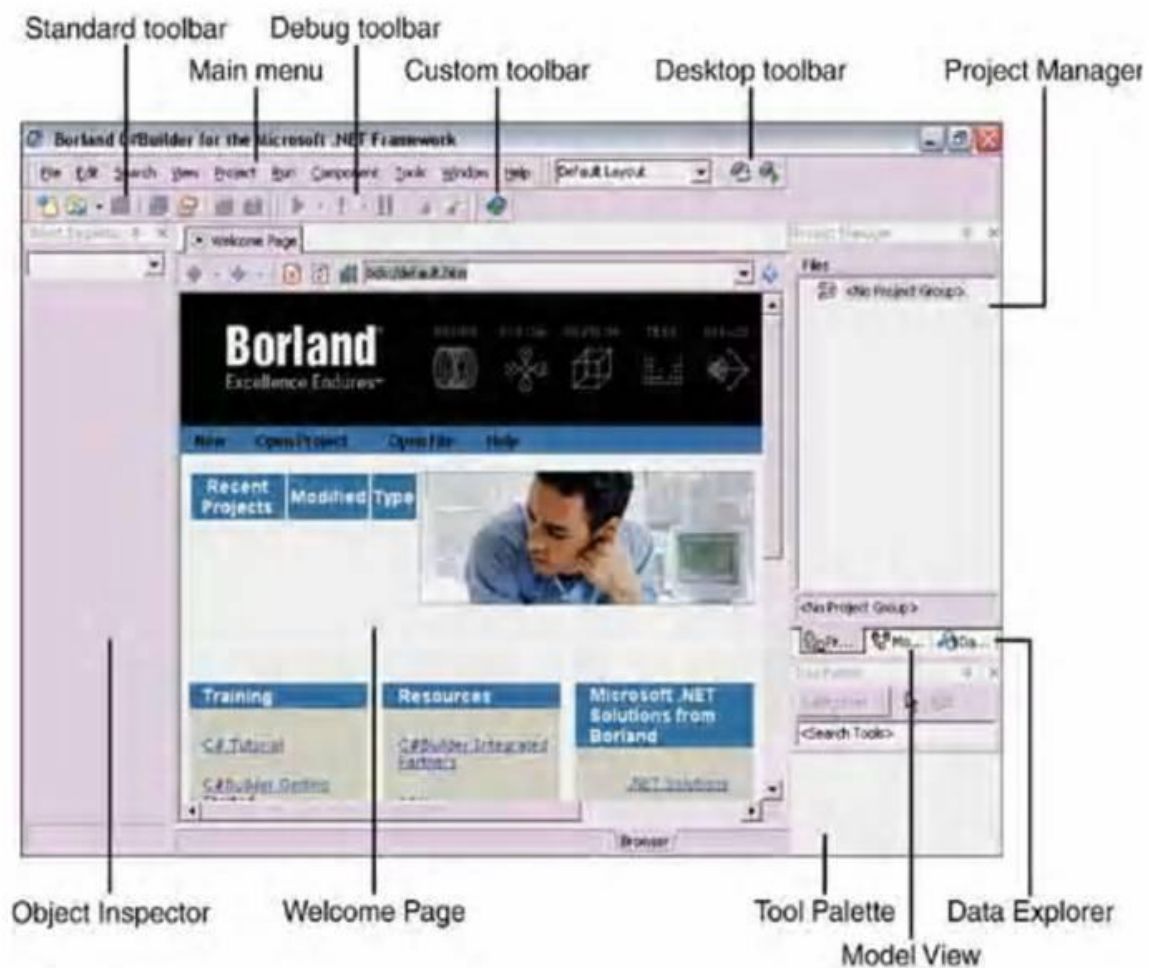
ü **C#Builder Borland Data Providers** برای محیط **.Net**، امکان دسترسی به پایگاه‌های داده **DB2 Universal**، **Oracle**، **InterBase** و **Microsoft SQL Server** را فراهم می‌نماید. البته در حال حاضر برخی از پایگاه‌های داده‌ای نظیر **Access** توسط این محیط پشتیبانی نمی‌شوند.

ü وجود ابزارهای متعددی جهت کار با پایگاه‌های داده، نظیر امکان برقراری ارتباط، ویرایش و اعمال تغییرات در پایگاه داده و همچنین امکان اجرای دستورات **SQL**.

ü **NET Menu Designers**. امکان تولید و ویرایش سریع منوها را برای شما فراهم می‌نماید تا بتوانید به سادگی منوهای لازم برای فرم‌های ویندوز خود را تولید و ویرایش نمایید.

نگاهی بر C#Builder

C#Builder محیطی کاملاً مجهز جهت ساخت و تولید برنامه‌ها تحت Net، بوسیله زبان برنامه‌نویسی C# است. این محیط شامل یک محیط گرافیکی پیشرفته است که دارای ابزارهای متعددی از قبیل ویراستارها (Editors)، ویزاردها و عیب‌یاب (Debugger) می‌باشد. پس از نصب و راه‌اندازی C#Builder صفحه‌ای همانند شکل 1 ظاهر می‌گردد.



شکل 1 صفحه ابتدایی آغاز C#Builder

صفحه اصلی و اولیه

مرکز صفحه، در شکل 1، صفحه شروع و آغاز C#Builder است. این صفحه شامل قسمتهایی بشرح زیر است: ساخت پروژه، باز کردن پروژه، فایل راهنمای محیط، پروژه‌های اخیر، آموزش، منابع و راه‌حل‌های Net. سه گزینه ساخت

پروژه، باز کردن پروژه و راهنما، لینک‌هایی به همین گزینه‌ها در منو و میله ابزار (**Toolbar**) هستند. با کلیک بر روی **New**، پنجره‌ای جهت انتخاب ویزاردی برای تولید پروژه‌ای جدید باز می‌شود. با کلیک بر روی **open** می‌توانید پروژه یا فایلی را انتخاب کرده و باز نمایید.

لینک **Help**، پنجره راهنما را باز خواهد کرد. ساخت یک پروژه جدید در قسمت بعدی مفصلاً توضیح داده خواهد شد. پس از ساخت یک پروژه جدید، نام این پروژه در لیست پروژه‌های اخیر (**Recent Projects**) وارد می‌شود و در هنگام شروع برنامه در دفعه‌های بعدی در این لیست نمایش داده می‌شود. قسمتهای آموزش (**Training**)، منابع (**Resources**) و راه‌حل‌های **Net**، لینک‌هایی جهت دسترسی به اطلاعاتی بیشتر هستند.

منوها و میله/ابزارها (**Menus & Toolbars**)

منوی اصلی محل دسترسی به تمامی امکانات موجود در **C#Builder** است. منوهای از پیش تعیین شده در **C#Builder** شامل منوهای استاندارد، منوهای عیب‌یابی و منوهای **Desktop** است. اگر بر روی هر یک از گزینه‌های منوها، پوینتر ماوس را برای چند لحظه نگاه دارید، توضیح مختصری به همراه کلید میانبر (**short key**) آن گزینه به نمایش در خواهد آمد. میله ابزار استاندارد شامل گزینه‌هایی جهت عملیات بر روی فایل است. گزینه‌های مربوط به عیب‌یابی شامل گزینه‌هایی جهت اجرا و عیب‌یابی برنامه است. در بخش بعدی این فصل به توضیح درباره‌ی چگونگی شخصی‌سازی میله ابزار می‌پردازیم.

پنجره پروژه (**Project Windows**)

در قسمت راست تصویر، مدیر پروژه (**Project Manager**)، **Model View** و پنجره مرورگر داده (**Data Explorer**) قرار دارد. پنجره **Project Manager** دقیقاً کاری را انجام می‌دهد که از اسمش پیداست؛ امکان مدیریت پروژه‌ها را برای شما فراهم می‌کند. پنجره **Model View** این امکان را فراهم می‌کند تا روابط موجود در پروژه خود را بصورت مدل‌های **UML** به تصویر کشیده و پروژه را از دیدگاه سطح بالایی کنترل و نظاره نمایید. **Data**

Explorer حاوی اطلاعاتی درباره پایگاه داده‌های موجود برای برنامه شماست. پایگاه داده‌های نمایش داده شده در این پنجره بستگی به نسخه **C#Builder** شما دارد. برای نمونه، نسخه حرفه‌ای **C#Builder (Professional)** تنها از **Borland InterBase** و **MS SQL Server** پشتیبانی می‌کند در حالیکه در نسخه **Enterprise** این نرم‌افزار علاوه بر دو پایگاه داده فوق، از دو پایگاه داده **IBM DB2** و **Oracle 9i** نیز پشتیبانی می‌گردد. البته شرکت بورلند اعلام کرده است که به زودی برای پشتیبانی از پایگاه داده‌های دیگری نظیر **Access** نیز اقدام خواهد کرد.

Tool Palette

محل دسته‌بندی اشیاء است که در پروژه‌های بسیاری مورد استفاده قرار می‌گیرند. اهمیت و ویژگی **Tool Palette** نیز در آنست که اشیاء درون آن بسته به نوع برنامه‌ای که تولید می‌شود، تغییر می‌کنند. برای مثال، هنگام کار با کد برنامه، این محل حاوی تکه کدهای قابل استفاده در برنامه است. در هنگام کار با فرمهای ویندوز، کنترل‌های اجرایی و مولفه‌های رایج در فرمهای ویندوز در این محل قرار دارند. اگر مشغول کار با **ASP.Net** هستید، این قسمت حاوی کنترل‌های سرور مخصوص **ASP.Net**، کنترل‌های **HTML** و سایر مولفه‌های مربوط به فرمهای وب و اجزای **ASP.Net** می‌باشد. در بخش استفاده از **Tool Palette** این فصل، به معرفی بیشتر این محل می‌پردازیم.

Object Inspector

این قسمت شامل جدولی از ویژگی‌هایی است که به کنترل فعلی یا مولفه‌ای که در حال حاضر مورد استفاده است، مربوط می‌باشد. در شکل 1، این محل خالی است، زیرا هیچ پروژه‌ای در حال اجرا نبوده و هیچ کنترلی مورد استفاده نمی‌باشد. هنگامیکه مولفه یا کنترلی انتخاب می‌شود، ویژگی‌های آن در این محل ظاهر می‌شوند. هنگامیکه شما از طریق **Object Inspector** ویژگی یک مولفه یا کنترل را تغییر می‌دهید، کد مربوط به آن نیز تغییر می‌کند. این روش، روشی آسان و سریع جهت کد کردن واسطه‌های گرافیکی کاربر (**GUI**) است.

اجزای نشان داده شده در شکل 1 تنها اجزایی ابتدایی در **C#Builder** هستند و ما در بخشهای بعدی این سایت سایر اجزای محیط **C#Builder** را به تفصیل توضیح خواهیم داد. قسمت بعدی این قسمت چگونگی آغاز یک پروژه و روش شروع به کار یک پروژه را توضیح خواهد داد.

برخی از قابلیت‌های محیط **C#Builder**

در این قسمت به بررسی برخی از قابلیت‌های موجود در محیط **C#Builder** پرداخته شده است.

شخصی سازی میله ابزار ها (Toolbars Customizing)

استفاده از Tool Palette

استفاده از Code Insight

استفاده از ویراستار HTML

میله ابزارهای HTML

محیط طراحی

Editor HTML Tag

اضافه کردن ابزارهای جدید به محیط کار

ایجاد ماکرو

شخصی سازی میله ابزار (Customizing Toolbar)

به دلیل اینکه میله ابزار پیش فرضی که از سوی شرکت بورلند تعبیه شده، ممکن است برآورده کننده نیازهای شما نباشد، امکان شخصی سازی میله ابزار به کاربر داده شده است. شخصی سازی میله ابزار در محیط **C#Builder** بسیار آسان است. فقط کافی است بر روی میله ابزار کلیک راست کنید و منوی **Customize** را انتخاب نمایید تا پنجره شخصی سازی میله ابزار ظاهر گردد. شاخه **Command** را انتخاب کنید، از لیست طبقه بندی شده سمت چپ گزینه **Run** را انتخاب نمایید و سپس از قسمت **Commands** گزینه **Run Without Debugging** را انتخاب نمایید. (به شکل 1-7 رجوع کنید) فقط کافیست جهت اضافه کردن این گزینه آنرا **Drag-Drop** کرده و در محل مورد نظر در میله ابزار قرار دهید. هنگام باز شدن پنجره **Customize** می توانید هر گزینه ای را که مد نظرتان هست در میله ابزار وارد نمایید. دو شاخه دیگر در پنجره **Customize**، شاخه های **Toolbars** و **Options** هستند. (به شکل 1-7

رجوع کنید) شاخه **Toolbars** مشخص می‌کند که کدام میله ابزارها در **IDE** نمایش داده شود. شاخه **Options** مربوط به چگونگی نمایش دادن راهنماهای میله ابزار است.

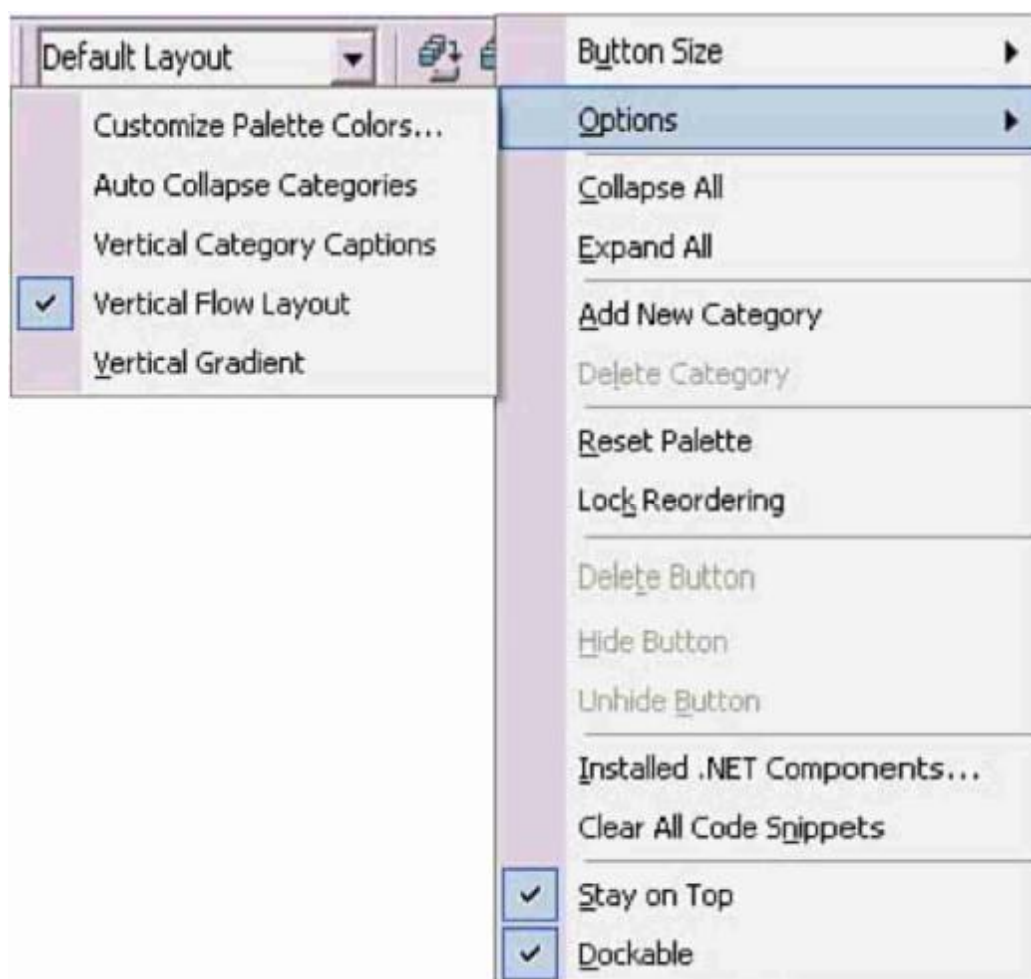


شکل 7-1 پنجره شخصی سازی میله ابزار

استفاده از *Tool Palette*

محتوای **Tool Palette** حساس به متن (**Context-Sensitive**) است و بستگی به نوع پروژه‌ای دارد که با آن کار می‌کنید. هنگام کار با کد، این بخش حاوی قطعه کدهایی است که می‌توان از آنها در برنامه استفاده نمود. هنگامیکه با کدهای **HTML** کار می‌کنید، این قسمت کنترل‌های **HTML** و هنگام کار با فرم‌های ویندوز، کنترل‌های ویندوز را در بر می‌گیرد. عناصر موجود در **Tool Palette** باید بصورت **drag-&-drop** وارد محیط کاری شوند و یا با دو بار کلیک بر روی آنها انتخاب گردند.

عناصر این قسمت جهت دسترسی آسانتر قابل طبقه‌بندی می‌باشند. تنها کافیست روی میله ابزار این قسمت کلیک راست کرده و جهت اضافه کردن یک دسته یا طبقه جدید **Add New Category** و جهت حذف آن **Delete Category** را انتخاب نمایید. (به شکل 8-1 توجه نمایید). **Tool Palette** دارای قابلیت فیلتر کردن عناصر نیز می‌باشد. بدین معنا که می‌توانید با تایپ چندین کلمه خاص عناصری که دارای این کلمات هستند را مشاهده نمایید. وجود فیلتر باعث می‌شود تا بتوانید از بین لیست طولانی **Tool Palette** عنصر یا عناصر مورد نظر خود را به سرعت پیدا نمایید.



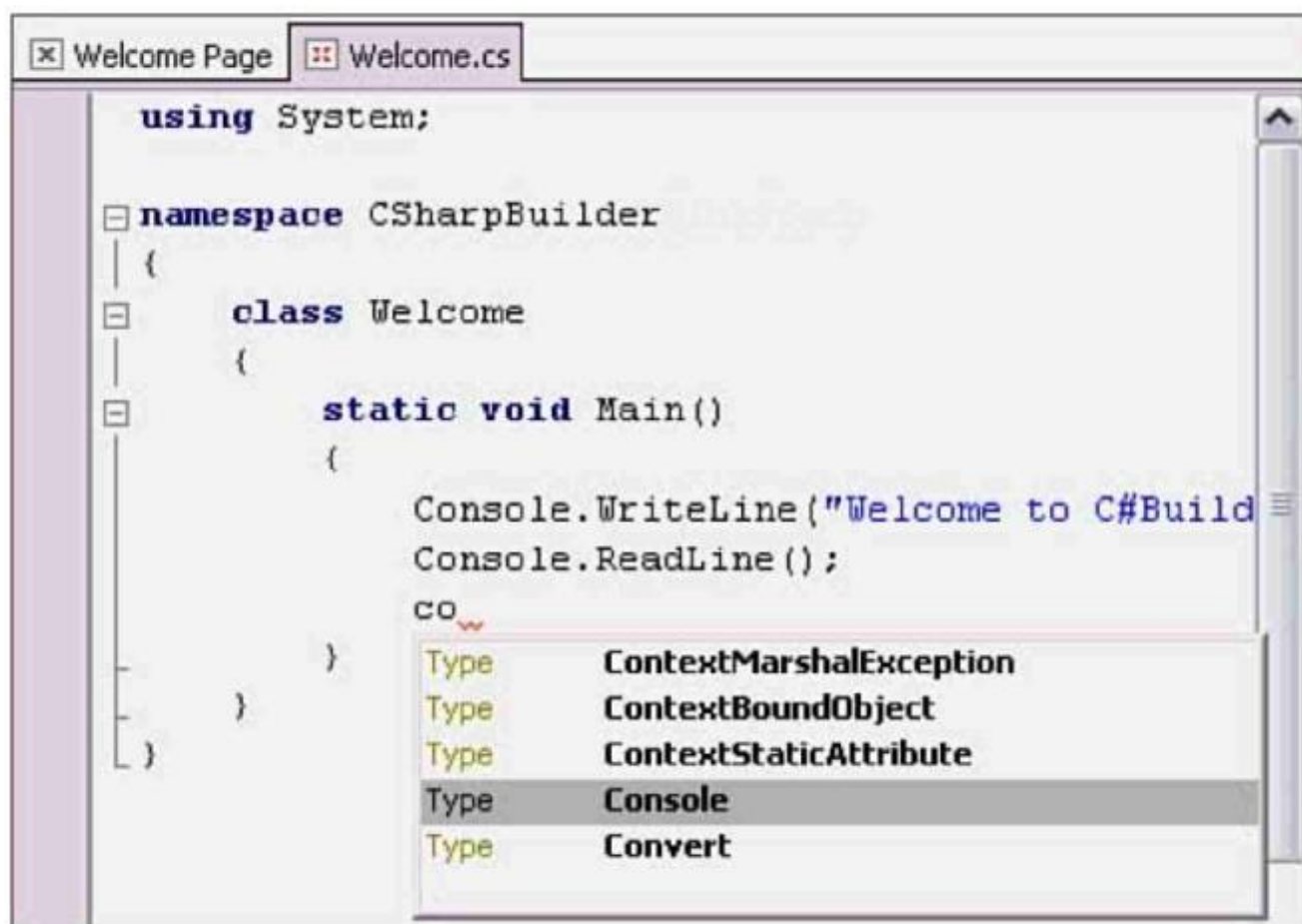
شکل 8-1 منوی متنی **Tool Palette**

درباره *Code Insight*

این ویژگی، باعث صرفه‌جویی در تایپ کدها می‌شود. آشنایی با چگونگی کار با این ویژگی، باعث می‌شود تا در زمان تایپ کدهای خود بسیار صرفه‌جویی نمایید.

برای شروع کار با **Code Insight**، در یک خط خالی **Ctrl+Spacebar** را فشار دهید. پس از این کار لیستی از نوعها ظاهر می‌گردد (به شکل 9-1 رجوع کنید). با تایپ هر کاراکتر از سوی شما، سایر قسمتهای باقی‌مانده فیلتر می‌گردد و باعث می‌شود تا بتوانید به عناصر دلخواه خود سریع دسترسی پیدا نمایید. بعنوان مثال، (همانطور که در شکل 9-1 نشان داده شده است) می‌خواهید عبارت **Console** را تایپ نمایید. در صورتیکه **Code Insight** فعال باشد، با تایپ کردن دو کاراکتر **Co** لیستی از کلماتی که با **Co** شروع می‌شوند ظاهر می‌گردد و شما می‌توانید به سرعت از درون این لیست **Console** را انتخاب کنید و وارد کد برنامه خود نمایید. پس از ظاهر شدن لیست، با انتخاب گزینه خاص و فشار دادن کلید **Enter**، گزینه مورد نظر وارد کد شما می‌گردد.

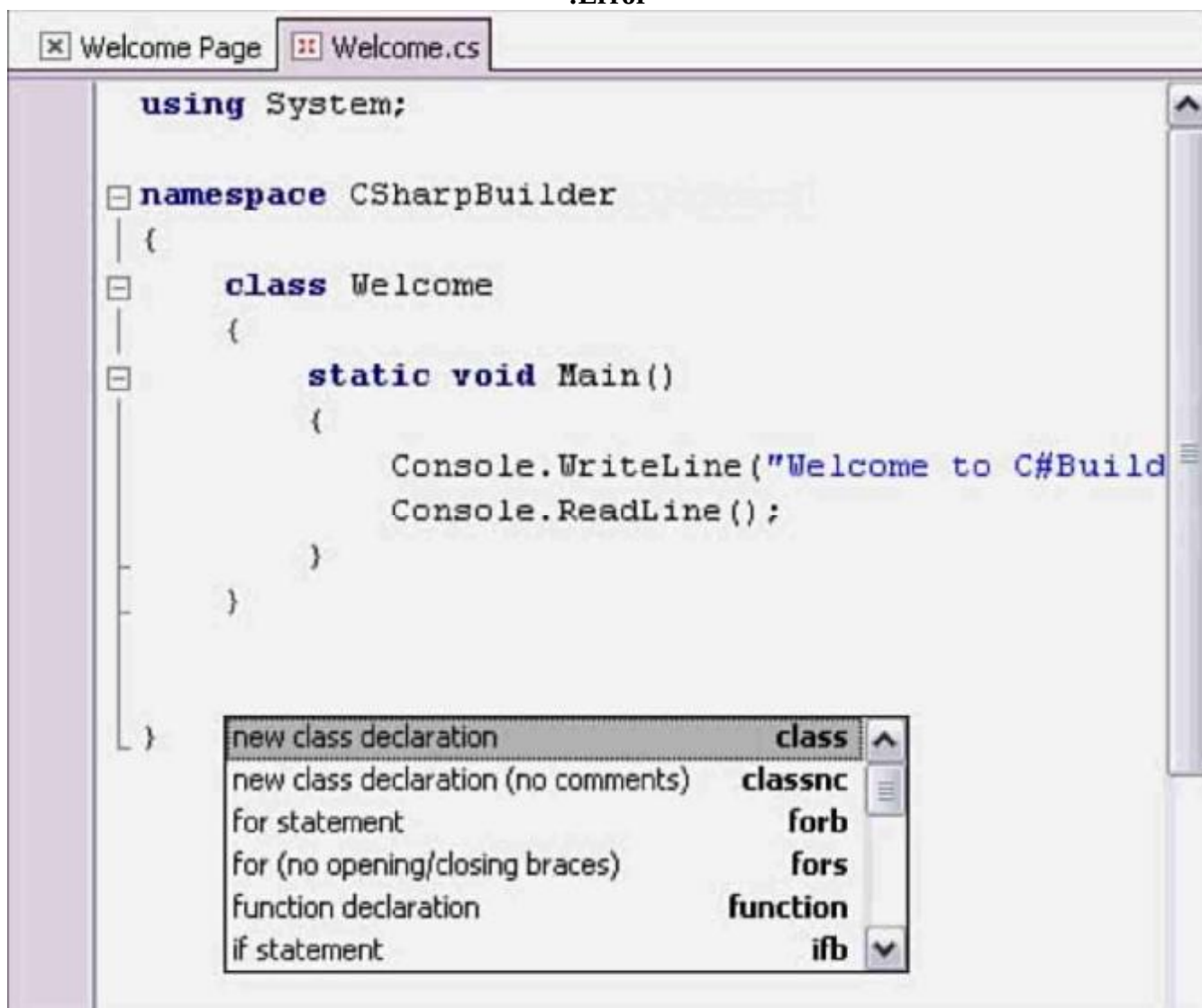
!Error



شکل 9-1 استفاده از **Code Insight** در حالت فعال بودن آن.

ویژگی و قابلیت دیگر **Code Insight** که با رفتن در یک خط خالی و فشار دادن **Ctrl+J**، لیستی از قطعه برنامه‌های متداول ظاهر می‌گردند. (به شکل 1-10 رجوع کنید). تایپ یک کاراکتر باعث می‌شود تا لیست به ابتدای محلی برود که با کاراکتر ورودی شما آغاز شده است. برای جستجو در لیست می‌توانید از کلیدهای **Up** و **Down** نیز استفاده نمایید.

!Error



شکل 1-10 استفاده از قابلیت **Snippets** در **Code Insight**

هنگامیکه عنصری از لیست انتخاب شود، کل بلوک کد آن در ویراستار کپی می‌شود. باید توجه شود که کدها در جایی که لازم است وارد شوند. بعنوان مثال، در داخل یک متد نیازی به وارد کردن کد مربوط به تعریف یک کلاس نیست.

برنامه‌نویسی در ویراستار **HTML**

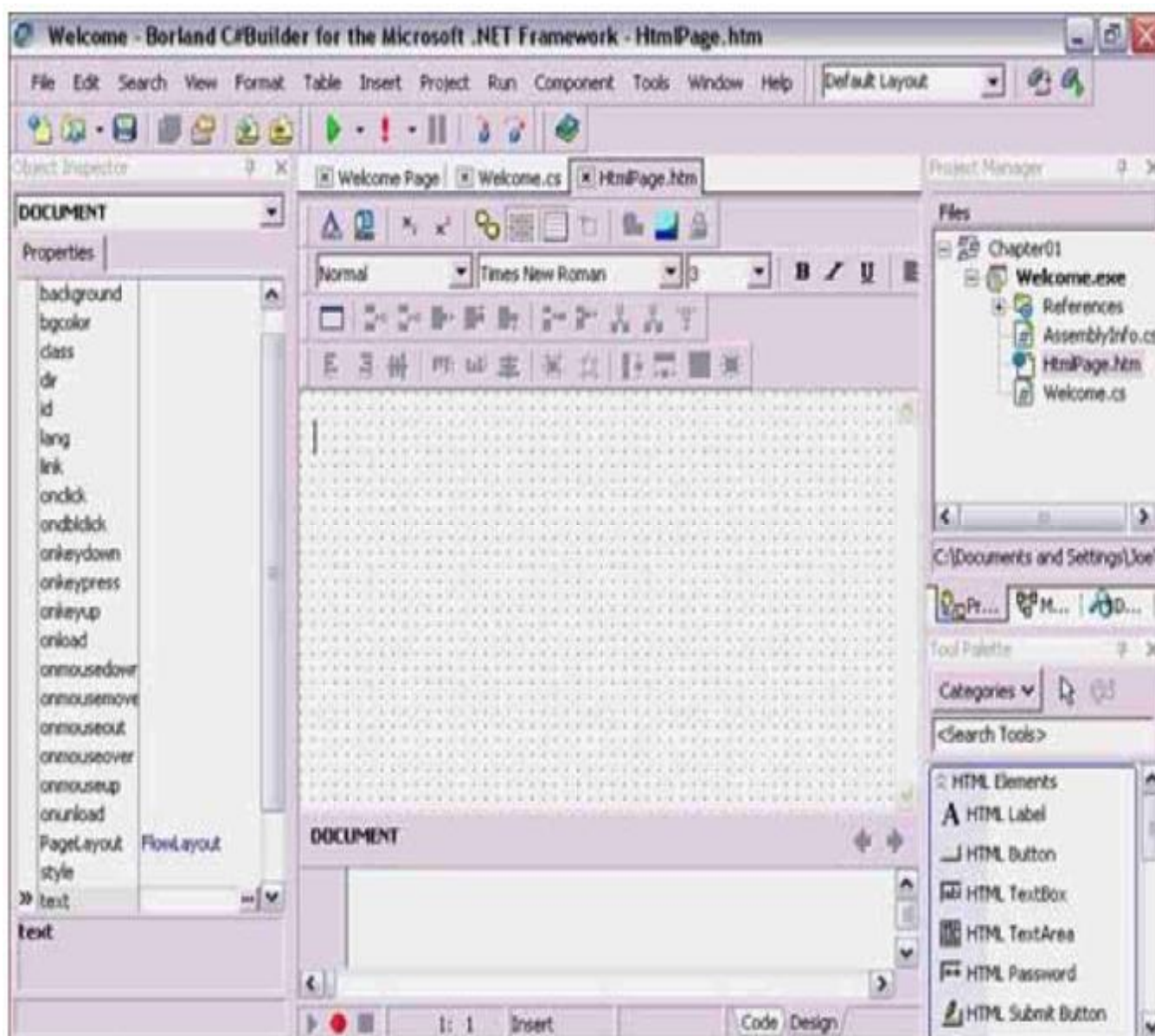
ویراستار HTML داخلی C#Builder دارای قابلیتهایی نظیر نوشتن کد، استفاده از tagها و ویرایش WYSIWYG می باشد. این امکانات، تسهیلات ویژه ای را جهت ساخت صفحات وب در اختیار برنامه نویس قرار می دهند.

برای آغار یک پروژه جدید HTML به منوی File، گزینه New رفته و سپس Others را انتخاب نمایید. (File>New>Others) در پنجره New Items از لیست Categories، گزینه Markup Files را انتخاب نمایید. سپس از لیست سمت راست HTML Page را انتخاب کرده و دکمه OK را کلیک نمایید.

این عمل شما باعث ایجاد یک فایل جدید HTML در حالت طراحی می گردد. (به شکل 1-11 رجوع نمایید) اگر پروژه دیگری در حال اجرا باشد، فایل جدید HTML شما به آن اضافه می گردد. اگر می خواهید تنها یک فایل HTML را تولید یا ویرایش کنید قبل از هر کاری تمامی پروژه های فعال خود را ببندید. همانطور که در شکل 1-11 نیز مشاهده می کنید پس از ایجاد یک فایل جدید HTML، در گوشه راست پائین تصویر Tool Palette فعال می گردد.

در این حالت Tool Palette حاوی کنترل های HTML است که می توانید آنها را در برنامه و محیط کار خود وارد نمایید. هنگامیکه مشغول ویرایش متن HTML هستید می توانید بین محیط ویراستار و محیط طراحی سوئیچ انجام داده و وارد هر یک از آنها گردید. محیط طراحی شامل سه قسمت: میله ابزار، محیط کار و ویراستار tag می باشد.

!Error



شکل 1-11 ویراستار HTML

میله ابزارهای HTML

میله ابزار پیش فرض جهت طراحی HTML شامل HTML Design, Align, HTML Table و HTML و

Format می باشد. همانند سایر میله ابزارها که در قسمتهای قبل گفته شد این قسمت نیز قابل شخصی سازی است. یکی

از حالت های از پیش تعیید شده در محیط HTML آنست که صفحه محیط طراحی بصورت شطرنجی و شبکه شبکه است

که شما می‌توانید با استفاده از دکمه **Show Grid** اقدام به فعال کردن یا غیر فعال کردن این حالت نمایید. عنصر مهم دیگر این میله ابزار دکمه **Inser Table** (وارد کردن جدول) است.

محیط طراحی

به محض اینکه بر روی سطح طراحی کلیک کنید، جدول ویژگیهای مربوطه در **Object Inspector** ظاهر می‌شوند. (به شکل 1-11 رجوع کنید) در قسمت سمت چپ بالای شکل، **Object Inspector** دیده می‌شود که لیستی تحت عنوان **Document** را در خود جای داده است. هدف از قرار دادن این لیست دسترسی سریع به عناصر **HTML** است. هر یک از خصیصه‌های این لیست نیز قابل ویرایش هستند. پس از ایجاد تغییر در این خصیصه با فشار دادن کلید **Enter** این تغییرات را در محیط طراحی مشاهده می‌کنید. پس از ایجاد این تغییرات کدهای **HTML** بصورت خودکار بروزرسانی (**Update**) می‌شوند و این مسئله را می‌توانید با کلیک بر روی قسمت **code** ملاحظه نمایید. هنگامیکه در حالت مشاده کد قرار می‌گیرید **Object Inspector** بصورت یک لیست خالی در می‌آید که پس از بازگشت به حالت طراحی مجدداً لیست عناصر آن به نمایش در می‌آیند.

ویرایش محیط طراحی بسادگی از طریق افزودن متن در کنار اشیا و اجزای **HTML** صورت می‌پذیرد. با استفاده از عناصر **Tool Palette** و میله ابزار، هر صفحه وبی را می‌توانید بصورت ویژوال طراحی نمایید. برای استفاده از عناصر موجود در **Tool Palette** تنها کافی است که دو بار بر روی عنصر مورد نظر کلیک نمایید تا آن عنصر بر روی محیط طراحی ظاهر شود و سپس آنرا در محل مناسبی که می‌خواهید قرار دهید. (این عمل بصورت **Drag-&Drop** نیز امکان پذیر است). پس از قرار دادن عنصر مورد نظر در محیط طراحی با انتخاب آن (یکبار روی آن کلیک کنید) کلیه مشخصات و ویژگیهای آن در **Object Inspector** به نمایش در می‌آید و می‌توانید آنها را ویرایش نمایید.

ویرایشگر tag های HTML

اگر دقت کرده باشید هنگامیکه عنصری را به محیط طراحی خود اضافه می‌نمایید و یا تغییری در صفحه بوجود می‌آورید، در قسمت پائینی صفحه طراحی، متنهایی تغییر می‌کنند. این قسمت ویرایشگر **tag** های **HTML** است که دارای قسمت

ویرایشگر **tag**، عنوان و **Zoom Control** است. قسمت عنوان، عنصری را نشان می‌دهد که اخیراً به محیط طراحی اضافه شده است. هربار که عنصر جدیدی را به محیط طراحی اضافه می‌کنید، عنوان و کدهای نمایش داده شده تغییر می‌کنند. ایجاد تغییرات در این **tag**ها باعث ایجاد تغییراتی در محیط طراحی و ویژگیهای عناصر می‌شوند. **Zoom Control** که همان پیکانه‌های سمت راست هستند، امکان پیمایش سلسله مراتبی در متن **HTML** را فراهم می‌کنند. این پیکانه‌ها تنها هنگامی فعال هستند که دارای چند سطح سلسله مراتبی باشیم. به عنوان مثال، هنگامیکه یک عنصر **Label** را به محیط طراحی اضافه می‌کنید، یک عنصر **HTML** با عنوان **** ایجاد می‌شود و بوسیله **Zoom Controls** می‌توانید بین نام **Label** و **tag** ایجاد شده **HTML** پیمایش داشته باشید. (پس از اضافه کردن **Label** در قسمت ویرایشگر **tag** عبارت **Label** دیده می‌شود که بیانگر کد **HTML** و **tag** مربوط به آن می‌باشد. در این حالت **Zoom Controls** غیر فعال است. حال در قسمت ویرایشگر کلیک کنید. مشاهده می‌کنید که این پیکانه‌ها فعال شده‌اند. با استفاده از آنها مشاهده می‌شود که متن نمایش داده شده در قسمت ویرایشگر تغییر می‌کند. شما می‌توانید از همین قسمت تغییراتی را در نام **Label** خود ایجاد کنید.)

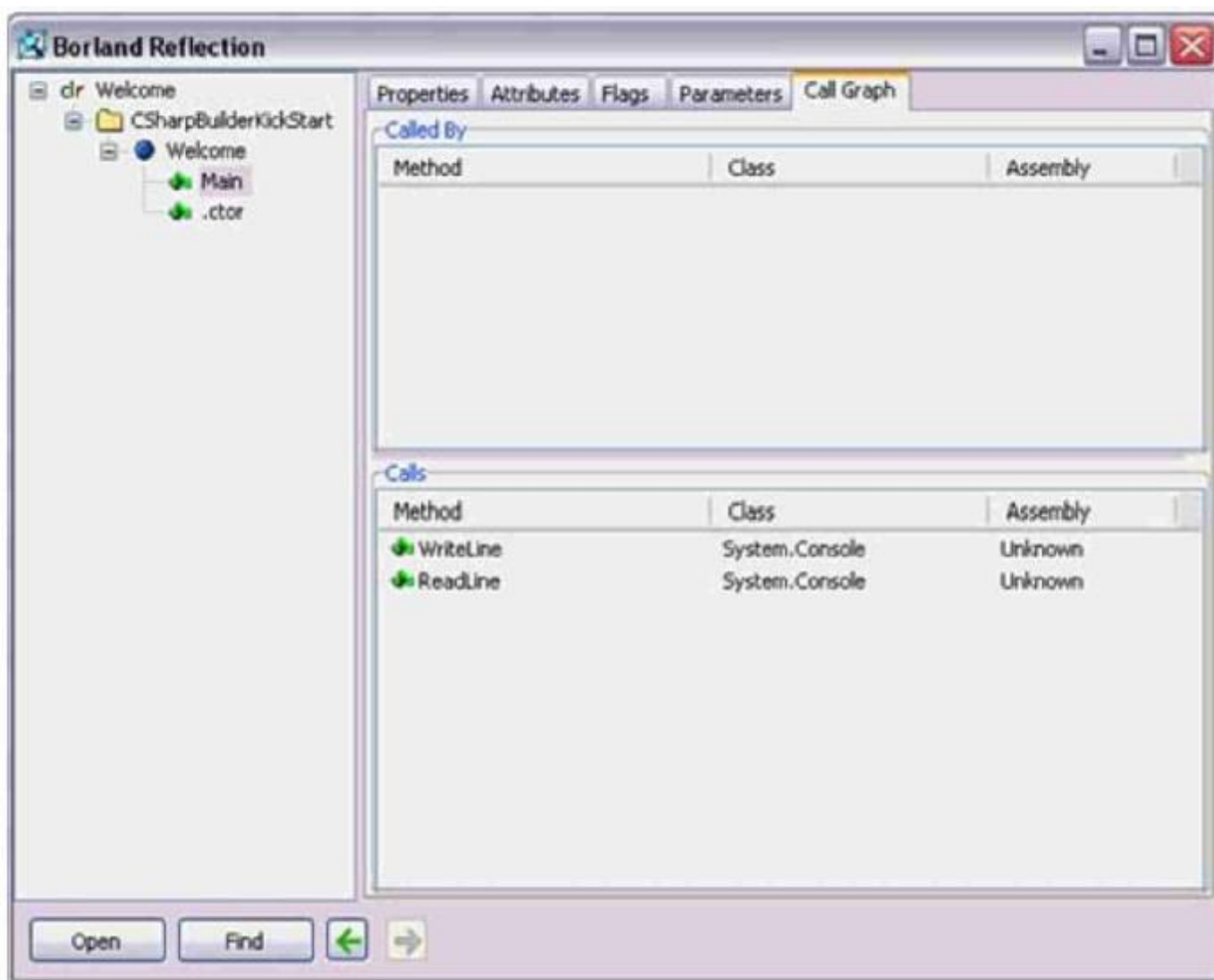
اضافه کردن ابزارها (Tools)

در کنار کار با **C#Builder** می‌توانید از برنامه‌ها و ابزارهای جانبی دیگری نیز استفاده نمایید. به منظور اینکه برای هر بار استفاده از این ابزارها مدتی را صرف پیدا کردن آنها در سیستم خود نکنید، **C#Builder** امکانی را جهت تسهیل دسترسی به این ابزارها فراهم کرده است. این امکان از طریق منوی **Tool** فراهم شده است. یکی از ابزارهایی که در این منو به صورت پیش فرض وجود دارد گزینه **Reflection** است.

Reflection ابزاریست که جزئیات داخلی اسمبلی را به کاربر نشان می‌دهد. استفاده از این ابزار بسیار آسان است. کافیست پس از انتخاب گزینه **Reflection** بر روی دکمه **Open** کلیک کنید و **dll** یا فایل **exe** مناسب را انتخاب نمایید. سپس پنجره‌ای حاوی اطلاعات مربوط به اسمبلی آن فایل برای شما به نمایش در خواهد آمد.

شکل 1-12 شمایی از این ابزار را پس از باز کردن یک فایل نشان می‌دهد. در این شکل اطلاعات مربوط به برنامه **Hello World** که در اولین مثال این کتاب ملاحظه نمودید، دیده می‌شود. در سمت چپ لیست نوعها و نام عناصر و در سمت راست اطلاعات مربوط به آنها نمایش داده می‌شود. با نشان دادن اطلاعات اسمبلی توسط **Reflection** اطلاعاتی در مورد برنامه‌هایی که فاقد مستندات هستند، می‌توان بدست آورد.

!Error



شکل 1-12 ابزار **Reflection**

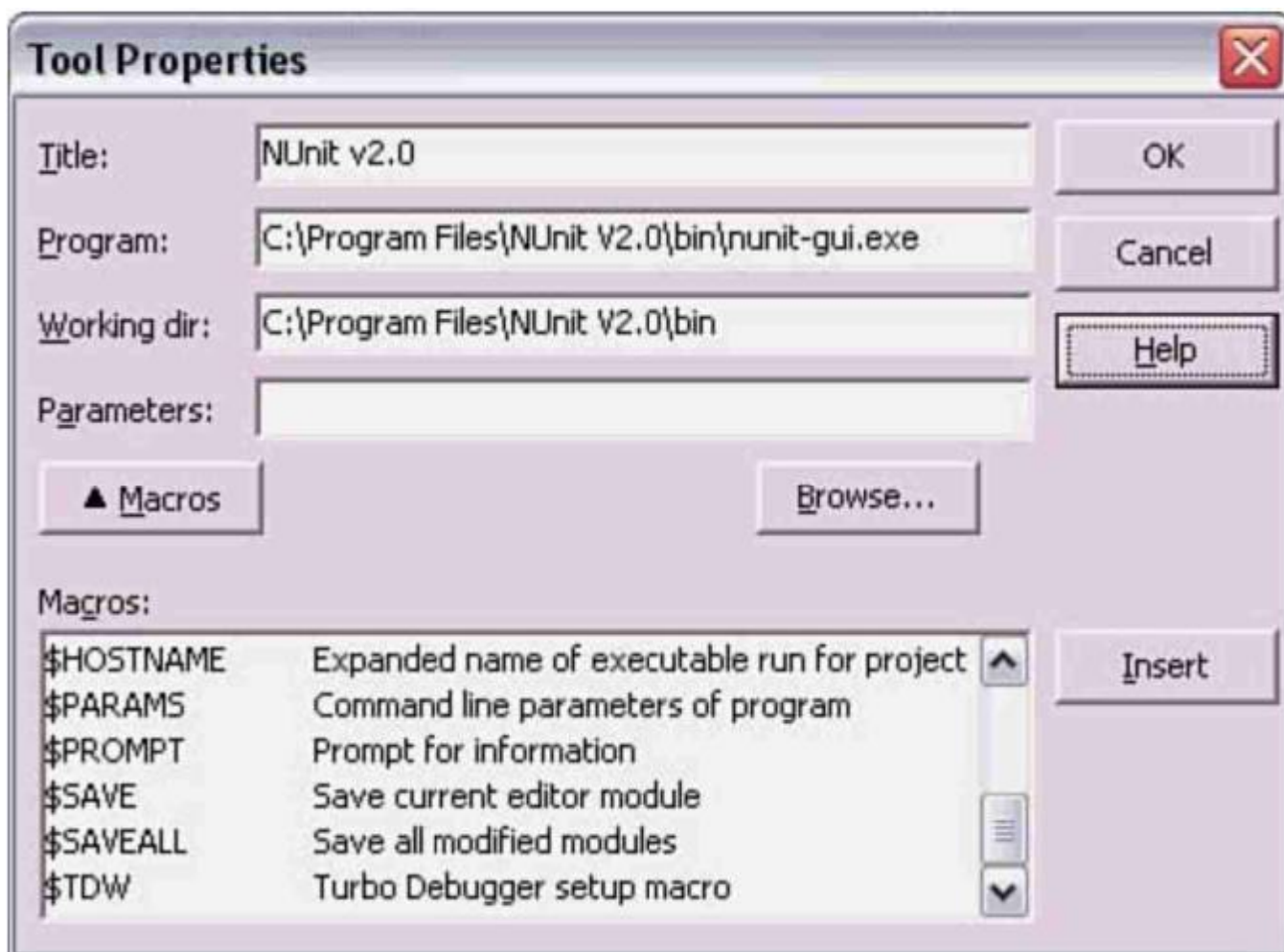
اگر می‌خواهید ابزاری را همانند **Reflection** در منوی **Tools** مشاهده کنید، گزینه **Configure Tools...** را از منوی **Tools** انتخاب نمایید و پس از باز شدن پنجره **Tools Option** (شکل 1-13) دکمه **Add** را کلیک نمایید. پس از این مراحل پنجره‌ای همانند شکل 1-14 نمایش داده می‌شود.



شکل 1-13 پنجره **Tools Options**

قسمت اول این پنجره عنوان ابزاریست که اضافه می‌کنید. با کلیک کردن بر روی **Browse** ابزار مورد نظر را که باید حتماً دارای یکی از سه پسوند **.pif**، **.com** و **.exe** باشد را انتخاب نمایید و سپس **OK** را کلیک نمایید. از این به بعد این ابزار از طریق منوی **Tools** قابل دسترسی خواهد بود. هنگام اضافه کردن ابزار، گزینه **Macro** در زیر پنجره دیده می‌شود. این گزینه جهت افزودن پارامترهای خاص به ابزار انتخابی مورد استفاده قرار می‌گیرد.

!Error



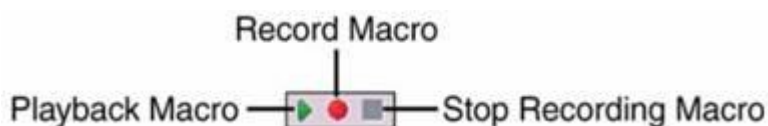
شکل 1-14 پنجره ویژگی‌های ابزار مورد نظر جهت اضافه کردن به منوی Tools

ایجاد ماکرو

C#Builder قابلیت‌های جهت ایجاد ماکرو را برای ضبط مراحل قابل استفاده مجدد در حین ایجاد یک برنامه را فراهم

می‌کند. دکمه‌های کنترل ماکرو در سمت چپ میله حالت (**Status bar**) قرار دارند. (شکل 1-15) جدول 1-3

کنترل‌های مختلف ماکرو را توضیح می‌دهد.



شکل 1-15 دکمه‌های کنترلی ماکرو

نام کنترل/توضیح	هدف
اجرای ماکرو/پیکان سبز رنگ	اجرای ماکرو
ضبط ماکرو/دایره قرمز رنگ	آغاز ضبط عملیات مربوط به ماکرو
توقف ضبط ماکرو/مربع سبز رنگ	خاتمه ضبط عملیات مربوط به ماکرو

جدول 1-3 کنترل‌های ماکرو و توضیح آنها

برای درک بهتر از این عمل و چگونگی ایجاد و کارکرد یک ماکرو به مثال زیر توجه نمایید. این ماکرو، یک ماکروی بسیار ساده است که مستندات و توضیحات XML را در بالای یک عنصر حاوی کد قرار می‌دهد.

1- پروژه جدیدی ایجاد کنید. (همانند آنچه در قسمتهای قبلی گفته شد) و یا فایل کلاس جدیدی را در پروژه فعلی

ایجاد نمایید. برای ایجاد فایل پروژه جدید به صورت زیر عمل نمایید **File>New>Others** و سپس در پوشه

New Files گزینه **Class** را انتخاب نمایید.

2- در این فایل کلاس، در بالای کلاس **Class1** کلیک کنید تا مکان نمای ماوس به آنجا هدایت شود. سپس یکبار

Enter را فشار دهید تا **public class Class1** یک خط به پایین برود.

3- شما در ابتدای یک خط خالی هستید. بر روی دکمه ضبط ماکرو را کلیک کنید.

4- عبارت **<summary>///** را تایپ کرده و **Enter** را فشار دهید.

5- عبارت **///** را تایپ کرده و **Enter** را فشار دهید.

6- عبارت **</summary>///** را تایپ کرده و **Enter** را فشار دهید.

7- بر روی دکمه توقف ضبط ماکرو را کلیک کنید.

مشاهده می‌کنید که دکمه کنترلی اجرای ماکرو فعال شده است. حال شما یک ماکرو ضبط کرده‌اید که می‌توانید از آن رد

هر نقطه از برنامه استفاده نمایید. برای دیدن چگونگی کارکرد ماکرو، پروژه‌ای را باز کنید و در هر جایی که می‌خواهید در

متن برنامه کلیک کنید تا مکان نمای ماوس به آنجا منتقل شود. حال بر روی دکمه اجرای ماکرو کلیک کنید. مشاهده

می‌کنید که عبارات تایپ شده شما عیناً در اینجا قرار می‌گیرند. توجه کنید که ضبط ماکرو تنها به فشردن شدن کلیدهای

صفحه کلید حساس است و آنها را ثبت می‌کند. یعنی هنگامیکه یک ماکرو ضبط می‌کنید، تمام فعالیت‌هایی که با صفحه کلید انجام می‌دهید (نه با ماوس) پست شده و هنگام اجرای ماکرو عیناً در محل مورد نظر اجرا می‌شوند. بعنوان یک تمرین مراحل زیر را انجام دهید و سپس اجرای آنرا در هر نقطه از برنامه تست نمایید. فرض کنید هدف ما از ضبط این ماکرو، افزودن یکسری توضیحات به ابتدای هر برنامه یا کلاس است.

- 1- فایل کلاس جدیدی ایجاد کنید.
- 2- مکان نما را به انتهای سطر اول یعنی **using System;** ببرید.
- 3- دکمه ضبط ماکرو را کلیک کنید.
- 4- با استفاده از دکمه **Home** بر روی صفحه کلید به ابتدای خط آمده و **Enter** را فشار دهید.
- 5- مکان نما را یک خط بالا برده و متن زیر را تایپ کنید.

```
//comment  
//using Macro  
//C# Builder
```

- 6- سپس دکمه توقف ضبط ماکرو را کلیک نمایید.

پس از ضبط شدن این ماکرو، یک فایل کلاس جدید ایجاد کنید و مکان نما را به انتهای سطر اول ببرید و سپس دکمه اجرای ماکرو را کلیک نمایید. ملاحظه می‌کنید که پس از اجرای ماکرو، خط اول یک سطر به پائین آمده و عبارات شما در بالای برنامه نوشته می‌شوند.

توجه: مجدداً یادآور می‌شوم که ماکرو تنها به فشردن دکمه‌های صفحه کلید حساس است و فعالیت‌های صفحه کلید رت در حین ضبط ماکرو ثبت می‌نماید. یعنی اعمالی که بوسیله ماوس انجام می‌شوند در ماکرو ذخیره نمی‌شوند.