

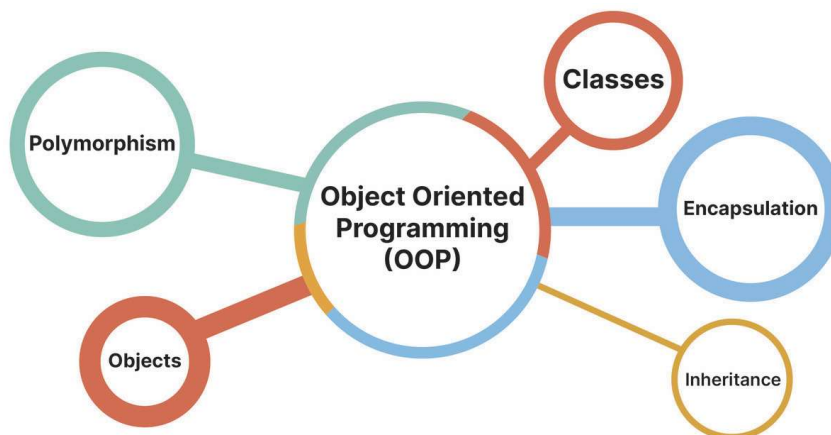
بسمه تعالی



دانشگاه جامع علمی کاربردی

برنامه نویسی شی گرا

Object Oriented Programming



مدرس: مهندس نوید همراهی

n.hamrahi@gmail.com

www.navidhamrahi.ir

جهت دریافت جزوه و اطلاع از روند کلاس به وب سایت بالا

بخش « دانشگاه مجازی » مراجعه فرمایید.

درس اول: آغاز کار با C#

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

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

لیست ۱-۱، 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 ۱-۱ دارای ۴ پارامتر اصلی است، اعلان Namespace، کلاس، متد Main() و یک دستور زبان C#. در همین جا باید به یک نکته اشاره کنم، برای زبان C# همانند بیشتر زبانهای برنامه سازی 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 می آیند. کامپایلرهای Visual محیطی همانند ویندوز دارند که با دارا بودن محیط گرافیکی و ابزارهای خاص، برنامه نویسی را در امر برنامه سازی کمک می کنند. از نمونه های هر یک از کامپایلرها، می توان به Microsoft C# Command Line 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 می شود. من سعی می کنم در آینده به توضیح محیط Visual C# و Visual Studio.Net بپردازم. اما فعلاً برای اجرای برنامه ها می توانید 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 برنامه ها انتخاب نمایید.

برای این درس، فعلاً به توضیحات بیشتر درباره محیط ویژوال نمی پردازم اما در آینده به توضیح کامل محیط Visual Studio.Net خواهیم پرداخت.

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

پس از اجرای برنامه، کامپایلر برای شما یک فایل قابل اجرا (Executable) تحت نام Welcome.exe تولید می کند. 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 می شود 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 می شود تا برنامه منتظر دریافت یک ورودی از کاربر بماند، که در این حالت شما می توانید خروجی برنامه خود را دیده و سپس با زدن کلید **Enter** برنامه را خاتمه دهید.

نکته دیگری که در مورد زبان برنامه نویسی 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 می شود.

اعلان **Namespace** به سیستم اعلان می نماید که شما از توابع کتابخانه ای **System** جهت اجرای برنامه ها خود استفاده می نمایید. دستوراتی مانند **WriteLine** و **ReadLine** جزو توابع کتابخانه ای **System** می باشند. اغلب دستورات و توابع مهم و کلیدی استفاده از کنسول ورودی/خروجی در این کتابخانه می باشد. 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 داده ها (متغیرها) و متدها جهت اجرای برنامه است. یک کلاس، جزو محدود عناصر زبان C# است که بوسیله آن می توان به ایجاد یک شی (Object) از قبیل واسط ها (Interfaces) و ساختارها (Structures)، پرداخت. توضیحات بیشتر در این زمینه در درس های آینده ذکر می شوند. در این برنامه کلاس هیچ داده و متغیری ندارد و تنها شامل یک متد است. این متد، رفتار (Behavior) این کلاس را مشخص می کند.

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

قبل از کلمه **Main()** کلمه دیگری با عنوان **static** آورده شده است. این کلمه در اصطلاح **Modifier** می گویند. استفاده از **static** برای متد **Main()** بیان می دارد که این متد تنها در در همین کلاس قابل اجراست و هیچ نمونه ای (Instance) دیگری از روی آن قابل اجرا نمی باشد. استفاده از **static** برای متد **Main()** الزامی است زیرا در ابتدای آغاز برنامه هیچ نمونه ای از هیچ کلاس و شی ای موجود نمی باشد و تنها متد **Main()** است که اجرا می شود. (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 برنامه نویسی شی گرا (OOP) هستند. سعی می کنم در درس های آینده به توضیح این مفاهیم نیز پردازم، ولی فعلاً در همین حد کافی می باشد).

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 نمی گرداند یا به عبارت بهتر خروجی ندارد. همچنین هر متد می تواند دارای پارامترهایی نیز باشد که لیست پارامترهای آن در داخل پرانتزهای جلوی آن قرار می گیرد. برای سادگی کار در این برنامه متد ما دارای هیچ پارامتری نیست ولی در ادامه همین درس به معرفی پارامترها نیز می پردازم.

متد **Main()** رفتار و عمل خود را بوسیله **Console.WriteLine(...)** مشخص می نماید. **Console** کلاسی در **System** است و **WriteLine()** متدی در کلاس **Console**. در زبان C# از اپراتور "." (نقطه) جهت جداسازی زیرروتین ها و زیرقسمتها استفاده می کنیم. همانطور که ملاحظه می کنید چون **WriteLine()** یک متد درون کلاس **Console** است به همین جهت از "." جهت جداسازی آن استفاده کرده ایم.

در زبان C#، برای قرار دادن توضیحات در کد برنامه از **//** استفاده می کنیم. بدین معنا که کامپایلر در هنگام اجرای برنامه توجهی به این توضیحات نمی کند و این توضیحات تنها به منظور بالا بردن خوانایی متن و جهت و کمک به فهم بهتر برنامه قرار می گیرند.

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 می گیرند به عنوان توضیحات (Comments) در نظر گرفته می شوند.

تمامی دستورات (Statements) با ";"، سمی کولون، پایان می یابند. کلاس ها و متدها با { آغاز شده و با } خاتمه می یابند. تمامی دستورات بین { } یک بلوک را می سازند.

بسیاری از برنامه ها از کاربر ورودی دریافت می کنند. انواع گوناگونی از این ورودی ها می توانند به عنوان پارامتری برای متد Main() در نظر گرفته شوند. لیست ۱-۲ برنامه ای را نشان می دهد نام کاربر را از ورودی دریافت کرده و آن را بر روی صفحه نمایش می دهد. این ورودی به صورت پارامتری برای متد Main() در نظر گرفته شده است.

لیست ۱-۲ : برنامه ای که ورودی را از کاربر، بعنوان پارامتر Main() دریافت می کند.

```
// 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 می کند 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 ۱-۲ مشاهده می نمایید، پارامتر متد Main() با عنوان args مشخص شده است. با استفاده از این نام در داخل متد می توان آن استفاده نمود. نوع این پارامتر از نوع آرایه ای از نوع رشته (string[]) در نظر گرفته شده است. انواع (types) و آرایه ها را در درس های بعدی بررسی می کنیم. فعلاً بدانید که آرایه رشته ای جهت نگهداری چندین کاراکتر مورد استفاده قرار می گیرد. [] مشخص کننده آرایه هستند که مانند یک لیست عمل می کنند.

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

مجدداً به آرگومان {0} بازگردیم. این آرگومان نشان می دهد 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 ۳ 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 ۱-۲ 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 ۳-۱ : 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 ۱-۲ 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 زمانی که 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 باز می 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 رشته ای (String) در زبان C#
- فراگیری چگونگی استفاده از آرایه ها (Arrays) در زبان C#

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

C# زبانی بسیار وابسته به انواع است، بطوریکه تمامی عملیاتی که بر روی داده ها و متغیرها در این زبان انجام می گیرد با دانستن نوع آن متغیر میسر می باشد. 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) می باشد. (به انواع Boolean از اینرو منطقی می گوئیم که تنها دارای دو حالت منطقی صحیح (True) و یا غلط (False) می باشند).

مثال ۱ – 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 می شوند. متغیرهای bool تنها می توانند یکی از دو مقدار true یا false را داشته باشند، یعنی همانند برخی از زبانهای برنامه سازی مشابه، مانند C و ++C، مقدار عددی نمی پذیرند، زیرا همانگونه که می دانید در این دو زبان هر مقدار عددی صحیح مثبت بغیر از صفر به عنوان true و عدد صفر به عنوان false در نظر گرفته می شود و در حقیقت نوع bool در این دو زبان نوعی integer می باشند. اما در زبان C# انواع bool یکی از دو مقدار true یا false را می پذیرند. 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 می‌دهد.

نوع	اندازه به بیت	رنج قابل قبول
sbyte	۸	۱۲۸- تا ۱۲۷
byte	۸	۰ تا ۲۵۵
short	۱۶	۳۲۷۶۸- تا ۳۲۷۶۷
ushort	۱۶	۰ تا ۶۵۵۳۵
int	۳۲	۲۱۴۷۴۸۳۶۴۷- تا ۲۱۴۷۴۸۳۶۴۸
uint	۳۲	۰ تا ۴۲۹۴۹۶۷۲۹۵
long	۶۴	۹۲۲۳۳۷۲۰۳۶۸۵۴۷۷۵۸۰۷- تا ۹۲۲۳۳۷۲۰۳۶۸۵۴۷۷۵۸۰۸
ulong	۶۴	۰ تا ۱۸۴۴۶۷۴۴۰۷۳۷۰۹۵۵۱۶۱۵

از این انواع برای محاسبات عددی استفاده می‌گردد. یک نوع دیگر را نیز می‌توان در این جدول اضافه نمود و آن نوع **char** است. هر چند شاید از نظر بسیاری از دوستانی که با زبانهای دیگر برنامه‌سازی کار کرده‌اند 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 ۶۵۵۳۵ دارد و اندازه آن نیز ۱۶ بیتی است، اما به جای نمایش دادن مقادیر عددی تنها می‌تواند بیان کننده یک کاراکتر باشد. در آینده در این مورد بیشتر توضیح خواهیم داد.

جدول زیر تمامی انواع عددی اعشاری زبان **C#** را نمایش می‌دهد.

نوع	اندازه به بیت	دقت	رنج قابل قبول
float	۳۲	۷ رقم	1.5×10^{-45} تا 3.4×10^{38}
double	۶۴	۱۵-۱۶ رقم	5.0×10^{-324} تا 1.7×10^{308}
decimal	۱۲۸	۲۸-۲۹ رقم دسیمال	1.0×10^{-28} تا 7.9×10^{28}

انواعی از نوع **floating point** هنگامی استفاده می‌شوند که محاسبات عددی به دقت‌های اعشاری نیاز داشته باشند. همچنین برای منظورهای 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 برنامه سازی نتایج بوسیله ایجاد یک سری عبارت تولید می گردند. عبارات از ترکیب متغیرها و عملگرها در دستورالعمل های یک زبان ایجاد می گردند. (توجه نمایید که عبارت معادل **expression** و دستورالعمل معادل **statement** می باشد که ایندو با یکدیگر متفاوت می باشند.) جدول زیر عملگرهای موجود در زبان **C#**، حق تقدم آنها و شرکت پذیری آنها را نشان می دهد.

شرکت پذیری	عملگر(ها)	نوع عمل
از چپ	x-- x++ a[x] f(x) x.y (x) new sizeof checked unchecked	عملیات ابتدایی
از چپ	(T)x --x ++x ~ ! - +	عملیات یکانی
از چپ	*/%	عملیات ضربی
از چپ	- +	عملیات جمعی
از چپ	<< >>	عمل شیفت
از چپ	< > <= >= is	عملیات رابطه ای
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 می شوند. شرکت پذیری از راست بدین معناست که تمامی محاسبات از راست به چپ صورت می گیرند. به عنوان مثال در یک عمل تساوی، ابتدا عبارات سمت راست تساوی محاسبه شده و سپس نتیجه به متغیر سمت چپ تساوی تخصیص داده می شود.

مثال ۲- 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 ۱ 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 ۲ می شود. (در مورد عملگر $--$ نیز چنین است.) پس با این توضیح می توان گفت که دو عملگر $++x$ و $x--$ ابتدا به عملوند خود یک واحد اضافه یا یک واحد از آن کم می کنند و سپس مقدار آنها را باز می گردانند.

در مثال ۲، مقدار متغیر **unary** در قسمت اعلان برابر با ۰ قرار گرفته است. هنگامیکه از عملگر $++x$ استفاده می کنیم، به مقدار متغیر **unary** یک واحد افزوده می شود و مقدارش برابر با ۱ می گردد و سپس این مقدار، یعنی ۱، به متغیر **preIncrement** تخصیص داده می شود. عملگر $x--$ مقدار متغیر **unary** را به ۰ باز می گرداند و سپس این مقدار را به متغیر **preDecrement** نسبت می دهد.

هنگامیکه از عملگر $x--$ استفاده می شود، مقدار متغیر **unary**، یا همان مقدار صفر، به متغیر **postDecrement** تخصیص داده می شود و سپس از مقدار متغیر **unary** یک واحد کم شده و مقدار این متغیر به ۱- تغییر می کند. سپس عملگر $++x$ مقدار متغیر **unary**، یعنی همان ۱-، را به متغیر **postIncrement** تخصیص می دهد و سپس یک واحد به مقدار متغیر **unary** می افزاید تا مقدار این متغیر برابر با ۰ (صفر) شود.

مقدار متغیر **bitNot** در هنگام اعلان برابر با صفر است. با استفاده از عملگر نقیض بیتی ($\sim$) (یا عملگر مکمل گیری) متغیر **bitNot** بعنوان یک بایت در نظر گرفته می شود و مقدار آن منفی یا نقیض می شود. 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 می گردد.

در این مثال به عبارت **(~bitNot)(sbyte)** توجه نمایید. هر عملی که بر روی انواع **short**، **ushort**، **byte** و **sbyte** انجام شود، مقداری از نوع **int** را باز می گرداند. بمنظور اینکه بتوانیم نتیجه دلخواه را به متغیر **bitNot** تخصیص دهیم باید از فرمت **operator (Type)** استفاده نماییم که در آن **Type** نوعی است می خواهیم نتیجه ما به آن نوع تبدیل شود و **operator** عملی است که بر روی متغیر صورت می پذیرد. به بیان دیگر چون می خواهیم 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 می شود 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 می خواهیم 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 می خواهیم نوع بزرگتر **int** را به (۳۲ بیتی) به نوع کوچکتر **sbyte** (۸ 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 (مثلا 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 داده های شما را تهدید می نماید. بحث در مورد مسائل پیچیده تر در مورد تبدیل انواع علامتدار و بدون علامت به یکدیگر در اینجا نمی گنجد و سعی می کنم 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 می آورد. در مثال بالا (مثال شماره ۲) 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 ۲ 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 ۳ - 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 ۳ 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 ۴- آرایه‌ها و عملیات بر روی آنها

```
using System;
class Array
{
    public static void Main()
    {
        int[] myInts = { 5, 10, 15 };
        bool[,] myBools = new bool[2, 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 ۴ 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 ۳ 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 ۴، اولین آرایه ما دارای ۳ عنصر است که مقادیر آنها به ترتیب برابر با ۵، ۱۰ و ۱۵ 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 ۱۵ عنصر تولید می‌نماید. یعنی این آرایه قادر است تا ۱۵ 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 ۳، ۵ و ۷ 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 بدهای آرایه را برابر با بزرگترین تعداد روز ماهها، یعنی ۳۱، تعریف نمایید. ولی چون تنها ۶ ماه دارای ۳۱ روز می‌باشند، برای ۶ 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 ۱۲ 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 ۱۲ عضو و بعد دوم آن ۳۱ 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 ۱۲ 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 می توان 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 می کنید که به هر ماه، تعداد عنصر مورد نیاز خود را تخصیص داده ایم. تنها باید به تفاوت اعلان آرایه های دنداندار با آرایه های 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++** 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 ۱۰ عضو دارد بزرگترین اندیس خانه هایش ۹ 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 ۳ و ستون ۷ 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 ۱۲ آرایه اشاره می کند اما فراموش نکنید چون اندیس آرایه از صفر آغاز می شود خانه شماره ۱۲ آرایه، سیزدهمین داده شما را در خود جای می دهد).

۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲
---	---	---	---	---	---	---	---	---	---	----	----	----

اگر شکل فوق را آرایه ای تک بعدی تصور نمایید، مشاهده می نمایید که خانه شماره ۵ 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 تک 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

درس سوم - دستورالعمل‌های کنترلی و شرطی

در این درس با دستورالعمل‌های کنترلی 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 ۱-۳ - 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 گروه
        if (myInt > 0)
        {
            Console.WriteLine("Your number {0} is greater than zero.", myInt);
        }
        // تصمیم‌گیری ساده و اجرای عمل بدون استفاده از دو گروه
        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() متد پایان
} //IfSelect کلاس پایان

```

برنامه ۱-۳ از یک متغیر **myInt** برای دریافت ورودی از کاربر استفاده می‌نماید، سپس با استفاده از یک سری دستورات کنترلی، که همان دستور **if** در اینجاست، عملیات خاصی را بسته به نوع ورودی انجام می‌دهد. در ابتدای این برنامه عبارت **Please enter a umber:** 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 پرانتزش 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 **{statements} if (boolean expression)** 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 گروه 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 گروه **{}** 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 نمی باشد. استفاده از بلوک تنها زمانی ضروری است که بخواهیم از چندین دستور استفاده نماییم.

دستور if-else

در بیشتر موارد از تصمیم گیری های دوگانه یا چندگانه استفاده می شود. در این نوع تصمیم گیری ها، دو یا چند شرط مختلف بررسی می شوند و در صورت **true** بودن یکی از آنها عمل مربوط به آن اجرا می گردد. سومین دستور **if** در این برنامه نشان دهنده یک تصمیم گیری دوگانه است. در این حالت در صورتیکه عبارت منطقی دستور **if** برابر با **true** باشد دستور بعد از **if** اجرا می شود و در غیر اینصورت دستور بعد از **else** به اجرا در می آید. در حقیقت در این حالت می گوئیم " 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 می گیرد و **statements** دستور یا دستوراتی است که اجرا می گردند.

دستور else if ... if یا if-else 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 تصمیم گیری چندگانه استفاده می نماییم. 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 ۱-۳ استفاده از **if** تودرتو را نشان می دهد. در این حالت نیز دستور با کلمه کلیدی **if** آغاز می گردد. شرطی بررسی شده و در صورت **true** بودن دستورات مربوط به آن اجرا می گردد. اما اگر مقدار این عبارت منطقی **false** بود آنگاه شرطهای فرعی دیگری بررسی می شوند. این شرطهای فرعی با استفاده از **else if** مورد بررسی قرار می گیرند. هر یک از این شرطها دارای عبارات منطقی مربوط به خود هستند که در صورت **true** بودن عبارت منطقی دستورات مربوط به آنها اجرا می گردد و در غیر اینصورت شرط بعدی مورد بررسی قرار می گیرد. باید توجه کنید که در ساختار **if** تودرتو تنها یکی از حالتها اتفاق می افتد و تنها یکی از شرطها مقدار **true** را بازی گرداند.

فرم کلی **if** تودرتو بشکل زیر است :

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

عملگرهای AND و OR (|| 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** بازی گرداند 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 می شود و دیگری **OR** معمولی که با **&&** نشان داده می شود. 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** باز می‌گرداند که هر دو عملوند **AND** دارای مقدار **true** باشند. یعنی در صورتیکه **myInt** هم بزرگتر از صفر باشد و هم کوچکتر از ۱۰، 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 ۲-۳ - 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.WriteLine("Type \"continue\" to go on or \"quit\" to stop: ");
        myInput = Console.ReadLine();
        // دستور switch به همراه متغیری از نوع رشته‌ای
        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 ۲-۳ 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: **sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enum** (نوع **enum** در مبحث جداگانه‌ای مورد بررسی قرار خواهد گرفت). در اولین دستور **switch** در مثال ۲-۳، 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 ۱، گزینه مورد نظر ما و دستور **Console.WriteLine(...)**، عمل مورد نظر است. در صورتیکه مقدار **myInt** برابر با عدد ۱ 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 **break** و دستور عملی در بین آنها، پشت سر هم قرار گیرند و دیگری در زمانی که 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 ۱، ۲ و ۳ دستور `Console.WriteLine("Your number is {0}.", myInt);` اجرا خواهد شد. یعنی اگر مقدار **myInt** برابر با هر یک از مقادیر ۱، ۲ و ۳ 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 ۲-۳، 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 **quit** یا **continue**) 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 بی‌مورد 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.

در این درس نحوه استفاده از دستورالعمل های کنترل حلقه در زبان C# را فرا خواهید گرفت. هدف این درس فهم و درک موارد زیر می باشد :

- ☐ حلقه while
- ☐ حلقه do-while
- ☐ حلقه for
- ☐ حلقه foreach
- ☐ مطالب تکمیلی درباره دستورالعمل break
- ☐ 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 حلقه ها در زبان C# آشنا خواهید شد. اولین دستوری که با آن آشنا می شوید 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.WriteLine("{0} ", myInt);
            myInt++;
        }
        Console.WriteLine();
    }
}
```

مثال ۴-۱ که در بالا ملاحظه می کنید، یک حلقه while ساده را نشان می دهد. این حلقه با کلمه کلیدی while آغاز شده و سپس به دنبال آن یک عبارت منطقی قرار می گیرد و مورد بررسی قرار می گیرد. تمامی دستورالعمل های کنترلی از یک عبارت منطقی بهره می گیرند 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 می گیرد تا چک شود آیا مقدارش از ۱۰ کوچکتر هست یا خیر. چون در ابتدای برنامه به این متغیر مقدار صفر تخصیص داده شده است، عبارت منطقی مقدار true را باز می گرداند و سپس بلوک قرار گرفته بعد از عبارت منطقی مورد اجرا قرار می گیرد.

درون بلوک while ابتدا مقدار متغیر myInt در کنسول نمایش داده می شود و سپس یک واحد به مقدار این متغیر افزوده می گردد. پس از اتمام بلوک while، عبارت منطقی مجدداً کنترل می شود و در صورتیکه این عبارت مقدار true بازگرداند، حلقه while مجدداً اجرا می شود. زمانی که عبارت منطقی مقدار false برگرداند، اجرا برنامه به اولین دستور بعد از بلوک while منتقل می شود. در این مثال اعداد صفر تا ۹ بر روی صفحه نمایش داده می شوند و سپس یک خط خالی چاپ شده و اجرای برنامه خاتمه می یابد.

حلقه بعدی که بسیار شبیه به حلقه while می باشد، حلقه do-while است.

حلقه *do-while*

ابتدا به مثال ۴-۲ توجه نمایید.

```

using System;

class DoLoop
{
    public static void Main()
    {
        string myChoice;
        do
        {
            // منوی 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 ۴-۲ نحوه استفاده از حلقه **do-while** را نشان می دهد. ساختار نوشتاری این دستور بصورت زیر است :

```

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

```

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

یکی از مصارف عمده حلقه **do** به جای حلقه **while**، مواردی است که می خواهیم یکسری دستورالعمل خاص، که آنها را درون بلوک **do** قرار می دهیم، حداقل یکبار اجرا شوند. در این مثال ابتدا یک منو برای کاربر نمایش داده می شود و سپس ورودی از وی دریافت می گردد. چون حلقه **while** عبارت منطقی خود در ابتدای اجرای حلقه بررسی می نماید، 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 ۴-۲ بیندازیم. در متد **Main()** متغیر **myChoice** را از نوع رشته ای تعریف نموده ایم. سپس یکسری دستورات را بر روی کنسول چاپ نموده ایم. 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 رشته ای قرار گیرد و از اینرو این ورودی را در **myChoice** قرار داده ایم. 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 می کنید، بری **default** نیز دستوری در نظر گرفته شده است که نشان می دهد مقدار ورودی معتبر نیست.

حلقه **for**

به مثال ۴-۳ 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.Write("{0} ", i);
        }
        Console.WriteLine();
    }
}

```

مثال ۴-۳ یک حلقه **for** را نشان می دهد. 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 ۳-۴ 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 ۳-۴ ما از دو دستور **if** درون حلقه **for** نیز استفاده کرده‌ایم. اولین دستور **if** بررسی می‌کند که آیا مقدار متغیر **i** برابر با ۱۰ 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 اسقناده از عملگر باقیمانده (%) بررسی می‌کند که آیا متغیر **i** بر ۲ بخش پذیر هست یا نه. در صورتیکه متغیر **i** بر ۲ بخش پذیر باشد، دستور **continue** اجرا می‌شود. پس از اجرای دستور **continue** از سایر دستورات حلقه **for** که بعد از **continue** قرار گرفته‌اند صرفه‌نظر 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 ۳-۴ 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.

حلقه **foreach**

به مثال ۴-۴ توجه کنید.

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 ۴-۴ 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 فقط خواندنی تعریف شده درون پرانتزهای **foreach** بازگردانده می‌شوند. تا زمانی که 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 ۴-۴، متغیری از نوع رشته با نام **person** برای نگهداری عناصری که از آرایه **names** خوانده می‌شود، تعریف کرده‌ایم. تا زمانی که 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 دستورات عمل‌های **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 :

- ☐ درک ساختار یک متد
- ☐ درک تفاوت بین متدهای استاتیک (static methods) و متدهای نمونه (instance)
- ☐ ایجاد نمونه جدید از اشیاء
- ☐ نحوه فراخوانی متدها
- ☐ درک چهار گونه متفاوت پارامترها
- ☐ 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، دستورهای از زبان C# هستند که بوسیله آنها عمل مورد نظر برنامه‌نویس انجام می‌شود و عملی است که یک متد آنرا انجام می‌دهد. مثال ۱-۵ 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 می‌شود
            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 موقوف می شود
Console.WriteLine();
Console.Write("Press Enter key to continue...");
Console.ReadLine();
Console.WriteLine();
} while (myChoice != "Q" && myChoice != "q");
// اجرای برنامه تا زمانی که کاربر بخواهد ادامه می یابد
}
string getChoice()
{
    string myChoice;
    // منویی را نمایش می دهد
    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 می کند
    myChoice = Console.ReadLine();
    Console.WriteLine();
    return myChoice;
}
}

```

برنامه مثال ۱-۵ دقیقاً همان برنامه درس ۴ است، با این تفاوت که در درس چهارم چاپ منو و دریافت ورودی از کاربر در متد **Main()** صورت می گرفت در حالیکه در این مثال، این اعمال در یک متد مجزا بنام **getChoice()** صورت می گیرد. نوع بازگشتی این متد از نوع رشته ای است. از این رشته در دستور **switch** در متد **Main()** استفاده می شود. همانطور که ملاحظه می نمایید، پرانتزهای متد **getChoice()** خالی هستند، یعنی این متد دارای پارامتر نیست، از اینرو هیچ اطلاعاتی به / از این متد منتقل نمی شود.

درون این متد، ابتدا متغیر **myChoice** را اعلان نموده ایم. هرچند نام و نوع این متغیر همانند متغیر **myChoice** موجود در متد **Main()** است، اما این دو متغیر، دو متغیر کاملاً مجزا از یکدیگر می باشند. هر دو این متغیرهای محلی (**Local**)

هستند، از اینرو تنها درون بلوکی که تعریف شده‌اند قابل دسترس 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 ۲-۵ توجه کنید.

```
using System;
```

```
class Address
```

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

```
class MethodParams
```

```
{
    public static void Main()
    {
        string myChoice;
        MethodParams mp = new MethodParams();
        do
        {
```

```

    // منویی نمایش داده شده و ورودی از کاربر دریافت می گردد
    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 پایان متد
// 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() پایان متد
// 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 ۵-۲، نمونه تغییر یافته مثال ۵-۱ 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 ۵-۲ کلاسی با نام **Address** با دو فیلد از نوع رشته تولید کرده ایم.

درون متد **Main()**، متد **getChoice()** را فراخوانی کرده ایم تا از کاربر ورودی دریافت کنیم و این ورودی در متغیر رشته ای **myChoice** قرار می گیرد. سپس متغیر **myChoice** را بعنوان آرگومان به متد **makeDecision()** ارسال نموده ایم. در اعلان **myDecision()**، همانطور که ملاحظه می نمایید، 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 می شود. در مورد پارامترهای مقداری (**value parameter**)، اصل مقدار متغیر یا پارامتر به پشته (**Stack**) کپی می شود. متغیرهایی که بصورت مقداری بعنوان پارامتر برای یک متد ارسال می شوند، همگی محلی بوده و تغییرات ایجاد شده بر روی آنها به هیچ وجه تغییری بر روی متغیر اصلی ایجاد نمی نماید.

دستور **switch** در متد **makeDecision()** برای هر **case** یک متد را فراخوانی می نماید. فراخوانی این متدها با آنچه در متد **Main()** دیدید مقداری متفاوت است. علاوه بر مرجع **mp**، در این فراخوانی ها 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 می شود و این مرجع همچنان به شیء اصلی درون **heap** نیز اشاره دارد چراکه آدرس شیء مورد نظر به متد کپی می شود. در مورد پارامترهای **ref**، هرگونه تغییری که بر روی متغیر محلی رخ دهد، همان تغییر بر روی متغیر اصلی نیز اعمال می گردد. امکان تغییر مرجع وجود ندارد و تنها شیء ای که مورد آدرس دهی واقع شده، می تواند تغییر پیدا نماید. پارامترهای مرجعی (**ref**) را می توان به عنوان عناصر ورودی/خروجی برای متد در نظر گرفت.

پارامترهای **out** در مواردی استفاده می شوند 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 می کاهد. در برنامه های عادی این مسئله چندان به چشم نمی آید، اما در برنامه های تحت شبکه که سرعت ارتباط و انتقال داده ها بسیار مهم است، این پارامترها ضروری می شوند.

متد **modifyAddress()** دارای پارامتری از نوع **out** است. پارامترهای **out** فقط به متد فراخواننده آن بازگشت داده می شوند. از آنجائیکه این پارامترها از متد فراخواننده هیچ مقداری دریافت نمی کنند و فقط درون متدی که به عنوان پارامتر به آن ارسال شده اند قابلیت تغییر دارند، از اینرو درون این متدهایی که به آنها ارسال می شوند، 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 می گردد. توجه نمایید که پارامترهای **out** می بایست قبل از دستور **return** درون متد مقدار دهی شده باشند.

یکی از ویژگیهای مفید زبان **C#**، وجود پارامترهای **params** است که بوسیله آنها می توان 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 دندانه دار (**Jagged Array**) باشند. در متد **makeDecision()** چهار متغیر رشته ای را به متد **viewAddresses()** ارسال نموده ایم که این متد پارامترهای خود را بصورت **params** دریافت می نماید. همانطور که ملاحظه می نمایید، تعداد متغیرهای ارسالی به متد می تواند متغیر باشد اما دقت داشته باشید که تمامی این متغیرها در یک آرایه تک بعدی قرار گرفته اند. درون متد **viewAddresses()** نیز با استفاده از دستور **foreach** تمامی عناصر موجود در این آرایه را نمایش داده ایم. پارامترهای **params** فقط متغیرهای ورودی دریافت می نمایند و تغییرات اعمال شده تنها بر روی متغیر محلی تاثیر می گذارد.

خلاصه

در این درس، با ساختار کلی یک متد آشنا شدید. فرا گرفتید 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 می باشد :

- ☐ آشنایی با **Namespace** در زبان **C#**
- ☐ چگونگی استفاده از هدایتگر **using** (*using directive*)
- ☐ چگونگی استفاده از هدایتگر **alias** (*alias directive*)
- ☐ 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 می شد. پس از مطالعه این درس، مطالب بسیاری درباره هدایتگر **using** فرا خواهید گرفت.

Namespace ها، المان های زبان **C#** هستند که شما را در سازماندهی کردن برنامه، کمک می کنند. پیاده سازی **Namespace** ها در کد برنامه بسیار مفید است چراکه از ایجاد مشکلات مربوط به استفاده مجدد کد، پیشگیری می نماید.

چگونگی ایجاد **Namespace**

به مثال زیر توجه نمایید.

```
// اعلان Namespace
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 ۱-۶ چگونگی ایجاد یک **Namespace** را نشان می دهد. در این مثال ما با قرار دادن کلمه کلیدی **namespace** در جلوی **csharp-persian** یک **Namespace** جدید ایجاد نمودیم. مثال ۲-۶، **Namespace** های تودرتو را نشان می دهد.

```
// اعلان Namespace
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 آییم، **Namespace** های تخصصی تر قرار می گیرند. این سیستم سلسله مراتبی بوسیله **Namespace** های تودرتو قابل پیاده سازی هستند. این پیاده سازی در مثال ۲-۶ نشان داده شده است. با قرار دادند کدهای خود در **Namespace** های فرعی می توانید کدهای خود را سازماندهی نمایید.

```
// اعلان Namespace
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 ۳-۶ 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 ۲-۶ است و از نظر پیاده سازی آسان تر از آن می باشد.

فراخوانی اعضای یک Namespace

```
// اعلان Namespace
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 ۴-۶ چگونگی فراخوانی اعضای **Namespace** نشان می‌دهد. در ابتدای مثال ۴-۶، 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 ۴-۶، یک **Namespace** دیگر بصورت **csharp-persian.tutorial** آورده شده است. کلاسهای **myExample2** و **myExample1** هر دو متعلق به یک **Namespace** می‌باشند، هر چند 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. هر چند کلاس **myExample2** خارج از محدوده ایست که متد **myPrint2** فراخوانی شده است، نیازی به آوردن نام **csharp-persian** برای این فراخوانی وجود ندارد، زیرا هر دو کلاس متعلق به یک **Namespace** است. 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*، به علت سادگی در امر یادگیری این مطلب، نامهای متفاوتی در نظر گرفته شده است. توجه نمایید که این دو متد به علت اینکه در کلاسهای متفاوتی قرار دارند، می توان نامهای یکسانی برایشان انتخاب نمود، و در اینصورت هیچگونه تداخلی بین ایندو ایجاد نخواهد شد.

استفاده از هدایتگر *using*

```
// اعلان Namespace
using System;
using csharp_station.tutorial;

// کلاس آغازین برنامه
class UsingDirective
{
    // آغاز اجرای برنامه
    public static void Main()
    {
        // Namespace عضوی از
        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 ۵-۶، 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 csharp-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 *csharp-persian.tutorial.myExample.myPrint()* می آوردیم.

```
// Namespace اعلان
using System;
using csTut = csharp-persian.tutorial.myExample; // alias

// کلاس آغازین برنامه
class AliasDirective
{
    // آغاز اجرای برنامه
    public static void Main()
    {
        // Namespace از عضویت
        csTut.myPrint();
        myPrint();
    }
    // متدی که ممکن است تولید ابهام نماید.
    static void myPrint()
    {
        Console.WriteLine("Not a member of csharp-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**) امکان پذیر است. همانطور که در مثال ۶-۶ 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 می توانند در خود داشته باشند که در زیر مشاهده می نمایید :

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 هر بار 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 استعاره ای (**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 می باشند :

- ☐ پیاده سازی سازنده ها (Constructors)
- ☐ درک تفاوت بین اعضای نمونه (Instance) و استاتیک (Static)
- ☐ آشنایی با تخریب کننده ها (Destructors)
- ☐ آشنایی با اعضای کلاسها

یک کلاس با استفاده از کلمه کلیدی **class** که بدنبال آن نام کلاس آمده باشد، اعلان می گردد و اعضای این کلاس درون {} اعلان می گردند. هر کلاس دارای سازنده ای می باشد که در هربار ایجاد نمونه ای جدید از آن کلاس، بصورت خودکار فراخوانی می گردد. هدف از سازنده، تخصیص دهی اعضای کلاس در زمان ایجاد نمونه ای جدید از کلاس است. سازنده ها دارای مقادیر بازگشتی نبوده و همواره نامی مشابه نام کلاس دارند.

مثال ۱-۷ نمونه ای از یک کلاس را نشان می دهد.

```
// Namespace اعلان
using System;

class OutputClass
{
    string myString;
    // سازنده
    public OutputClass(string inputString)
    {
        myString = inputString;
    }
    // متد نمونه
    public void printString()
    {
        Console.WriteLine("{0}", myString);
    }
    // تخریب کننده
    ~OutputClass()
    {
        // روتینی جهت آزادسازی برخی از منابع سیستم
    }
}

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

}

در مثال ۷-۱ 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 ۷-۱ 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، **Math** می‌باشد که از اعضای استاتیک بهره می‌برد. **Framework BCL** خود دارای کلاس **Math** می‌باشد که از اعضای استاتیک بهره می‌برد.

نوع دیگر سازنده‌ها، سازنده‌های استاتیک هستند. از سازنده‌های استاتیک جهت مقداردهی 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 مقداردهی 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 :

- ☐ پیاده‌سازی کلاسهای پایه (Base Class)
- ☐ پیاده‌سازی کلاسهای مشتق شده (Derived Class)
- ☐ مقدار دهی کلاس پایه از طریق کلاس مشتق شده
- ☐ فراخوانی اعضای کلاس پایه
- ☐ پنهان‌سازی اعضای کلاس پایه

ارث‌بری یکی از مفاهیم اساسی و پایه شی‌گرایی 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 ۸-۱ 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 توانائیهای **ParentClass** را دارد. از اینرو می‌توان گفت **ChildClass** یک **ParentClass** است. **ChildClass (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 ۸-۱ توجه نمایید. سازنده **ParentClass** قبل از سازنده **ChildClass** اجرا می‌گردد.

برقراری ارتباط کلاس مشتق شده با کلاس پایه
به مثال ۸-۲ 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 ۲-۸ 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 خطای 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 کلاسی 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؟
- پیاده سازی متد مجازی (Virtual Method)
- Override کردن متد مجازی
- استفاده از چند ریختی در برنامه ها

یکی دیگر از مفاهیم پایه ای در شی گرایی، چند ریختی (Polymorphism) است. با استفاده از این ویژگی، می توان برای متد کلاس مشتق شده پیاده سازی متفاوتی از پیاده سازی متد کلاس پایه ایجاد نمود. این ویژگی در جایی مناسب است که می خواهید گروهی از اشیاء را به یک آرایه تخصیص دهید و سپس از متد هر یک از آنها را استفاده کنید. این اشیاء الزاماً نباید از یک نوع شیء باشند. هر چند اگر این اشیاء بواسطه ارث بری به یکدیگر مرتبط باشند، می توان آنها را بعنوان انواع ارث بری شده به آرایه اضافه نمود. اگر هر یک از این اشیاء دارای متدی با نام مشترک باشند، آنگاه می توان هر یک از آنها را جداگانه پیاده سازی و استفاده نمود. در این درس با چگونگی انجام این عمل آشنا می گردید.

متد مجازی (Virtual Method)

using System;

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

مثال ۹-۱ کلاس *DrawingObject* را نشان می دهد. این کلاس می تواند بعنوان کلاسی پایه جهت کلاسهای دیگر در نظر گرفته شود. این کلاس تنها دارای یک متد با نام *Draw()* می باشد. این متد دارای پیشوند *virtual* است. وجود کلمه *virtual* بیان می دارد که کلاسهای مشتق شده از این کلاس می توانند، این متد را *override* نمایند و آنرا به طریقه دلخواه پیاده سازی 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 ۹-۲، سه کلاس دیده می شود. این کلاسها از کلاس *DrawingObject* ارث بری می کنند. هر یک از این کلاسها دارای متد *Draw()* هستند و تمامی آنها دارای پیشوند *override* می باشند. وجود کلمه کلیدی *override* قبل از نام متد، این امکان را فراهم می نماید تا کلاس، متد کلاس پایه خود را *override* کرده و آنرا به طرز دلخواه پیاده سازی نماید. متدهای *override* شده باید دارای نوع و پارامترهای مشابه متد کلاس پایه باشند.

پیاده سازی 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 ۹-۳ برنامه ای را نشان می دهد که از کلاسهای مثال ۹-۱ و ۹-۲ استفاده می کند. در این برنامه چند ریختی پیاده سازی 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 می دارد.

سپس آرایه *dObj* تخصیص دهی شده است. به دلیل رابطه ارث بری این عناصر با کلاس *DrawingObject*، عناصر *Line*، *Square* و *Circle* قابل تخصیص به این آرایه می باشند. بدون استفاده از این قابلیت، قابلیت ارث بری، برای هر یک از این عناصر باید آرایه ای جدا می ساختید. ارث بری باعث می شود تا کلاسهای مشتق شده بتوانند همانند کلاس پایه خود عمل کنند که این قابلیت باعث صرفه جویی در وقت و هزینه تولید برنامه می گردد.

پس از تخصیص دهی آرایه، حلقه *foreach* تک تک عناصر آنرا پیمایش می کند. درون حلقه *foreach* متد *Draw()* برای هر یک از اعضای آرایه اجرا می شود. نوع شیء مرجع آرایه *dObj*، *DrawingObject* است. چون متد *Draw()* در هر یک از این اشیاء *override* می شوند، از اینرو متد *Draw()* مربوط به هر یک از این اشیاء اجرا می شوند. 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 می شوند. 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 زبان های برنامه نویسی شی گرا است و از طریق آن می توان برای یک متد موجود در کلاس پایه، چندین پیاده سازی 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 می باشد :

- موارد استفاده از Property ها
- پیاده سازی 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 می نماید. **Property** ها علاوه بر اینکه از فیلدهای یک کلاس حفاظت می کنند، همانند یک فیلد قابل دسترسی هستند. بمنظور درک ارزش **Property** ها بهتر است ابتدا به روش کلاسیک کپسوله کردن متدها توجه نمایید.

مثال ۱-۱۰ : یک نمونه از چگونگی دسترسی به فیلدهای کلاس به طریق 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 ۱-۱۰ روش کلاسیک دسترسی به فیلدهای یک کلاس را نشان می دهد. کلاس **PropertyHolder** دارای فیلدی است تمایل داریم به آن دسترسی داشته باشیم. این کلاس دارای دو متد **getSomeProperty()** و **setSomeProperty()** می باشد. متد **getSomeProperty()** مقدار فیلد **someProperty** را باز می گرداند و متد **setSomeProperty()** مقداری را به فیلد **someProperty** تخصیص می دهد.

کلاس **PropertyTester** از متدهای کلاس **PropertyHolder** جهت دریافت مقدار فیلد **someProperty** از کلاس **PropertyHolder** استفاده می کند. در متد **Main()** نمونه جدیدی از شی **PropertyHolder** با نام **propHold** ایجاد می گردد. سپس بوسیله متد **setSomeProperty** مقدار **someMethod** از **propHold** برابر با ۵ می گردد و سپس برنامه مقدار **property** را با استفاده از فراخوانی متد **Console.WriteLine()** در خروجی نمایش می دهد. آرگومان مورد استفاده برای بدست آوردن مقدار **property** فراخوانی به متد **getSomeProperty()** است که توسط آن عبارت "Property Value : 5" در خروجی نمایش داده می شود.

چنین متد دسترسی به اطلاعات فیلد بسیار خوب است چرا که از نظریه کپسوله کردن شیءگرایی پشتیبانی می کند. اگر پیاده سازی **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 ساده تر پیاده سازی می گردد. به مثال زیر توجه نمایید.

مثال ۲-۱۰: 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 ۲-۱۰ چگونگی ایجاد و استفاده از ویژگیها (**Property**) را نشان می دهد. کلاس **PropertyHolder** دارای پیاده سازی 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** می باشد. ویژگی دارای دو **accessor** با نامهای **get** و **set** است. **get accessor** مقدار فیلد **someProperty** را باز می گرداند. **set accessor** نیز با استفاده از مقدار **value**، مقداری را به **someProperty** تخصیص می دهد. کلمه **value** که در **set accessor** آورده شده است جزو کلمات رزرو شده زبان **C#** می باشد.

کلاس **PropertyTester** از ویژگی **someProperty** مربوط به کلاس **PropertyHolder** استفاده می کند. اولین خط در متد **Main()** شیء از نوع **PropertyHolder** با نام **propHold** ایجاد می نماید. سپس مقدار فیلد **someProperty** مربوط به شیء **propHold** با استفاده از ویژگی **SomeProperty** به ۵ تغییر می یابد و ملاحظه می نمایید 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 می نماید. این عمل با استفاده از ویژگی **SomeProperty** شیء **propHold** صورت می گیرد.

ویژگیها را می توان 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 ۳-۱۰: 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 ۳-۱۰ چگونگی ایجاد یک ویژگی فقط خواندنی را نشان می دهد. کلاس **PropertyHolder** دارای ویژگی **SomeProperty** است که فقط **get accessor** را پیاده سازی می کند. این کلاس **PropertyHolder** دارای سازنده ایست که پارامتری از نوع **int** دریافت می نماید.

متد **Main()** از کلاس **PropertyTester** شیء جدیدی از **PropertyHolder** با نام **propHold** ایجاد می نماید. این نمونه از کلاس **PropertyHolder** از سازنده آن که مقداری صحیح را بعنوان پارامتر دریافت می کند، استفاده می کند. در این مثال این مقدار برابر با ۵ در نظر گرفته می شود. این امر باعث تخصیص داده شدن عدد ۵ به فیلد **someProperty** از شیء **propHold** می شود.

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 ۴-۱۰: 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 ۴-۱۰ 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** شی‌ای جدید از همین کلاس با سازندهٔ پیش فرض آن ایجاد می‌نماید. سپس با استفاده از ویژگی **SomeProperty** از شیء **propHold**، مقدار ۵ را به فیلد **someProperty** مربوط به شیء **propHold** تخصیص می‌دهد. در این حالت **set accessor** مربوط به ویژگی **SomeProperty** فراخوانی شده و مقدار ۵ را به فیلد **someProperty** تخصیص می‌دهد و سپس عبارت **"someProperty is equal to 5"** را در خروجی نمایش می‌دهد.

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 می‌گرفت ولی با استفاده از ویژگیها (**Property**) می‌توان به اجزای یک شیء همانند یک فیلد دسترسی پیدا کرد. ویژگیها را می‌توان 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 امکان‌پذیر می‌گردد.

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 ۱۱-۱ : 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 ۱۱-۱ نحوه پیاده سازی اندیکسر را نشان می دهد. کلاس **IntIndexer** دارای آرایه رشته ای بنام **myData** می باشد. این آرایه، عنصری خصوصی (**private**) است و کاربران خارجی (**external users**) نمی توانند به آن دسترسی داشته باشند. این آرایه درون سازنده (**constructor**) کلاس تخصیص دهی می گردد که در آن پارامتر **size** از نوع **int** دریافت می شود، از آرایه **myData** نمونه ای جدید ایجاد می گردد، سپس هر یک از المانهای آن با کلمه "empty" مقداردهی می گردد.

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 می نماید. همانطور که حتماً تا کنون دریافته اید پیاده سازی اندیکسر بسیار شبیه به پیاده سازی یک ویژگی (**property**) است. اندیکسر نیز دارای **accessor** های **set** و **get** است که دقیقاً همانند **property** عمل می کنند. همانطور که در اعلان این اندیکسر نیز مشاهده می شود، متغیری از نوع رشته ای را باز می گرداند.

در متد **Main()** شیء جدیدی از **IntIndexer** ایجاد شده است و مقادیری به آن افزوده می شود و سپس نتایج چاپ می گردند. 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 آرایه ها در اغلب زبانهای برنامه سازی رایج است ولی زبان **C#** چیزی فراتر از آنرا نیز پشتیبانی می کند. در **C#** اندیکسرها را می توان با چندین پارامتر تعریف کرد و هر پارامتر می تواند از نوع خاصی باشد. پارامترهای مختلف بوسیله کاما از یکدیگر جدا می شوند. پارامترهای مجاز برای اندیکسر عبارتند از : **integer**، **enum** و **string**. علاوه بر آن، اندیکسرها قابل سرریزی (**Overload**) هستند. در مثال ۲-۱۱ تغییراتی در مثال قبل ایجاد کرده ایم تا برنامه قابلیت دریافت اندیکسرها را نیز داشته باشد.

سرریزی اندیکسرها

مثال ۲-۱۱: اندیکسرها سرریز شده (**Overloaded Indexers**)

```

using System;
/// <summary>
/// پیاده سازی 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 ۱۱-۲ نحوه سرریز کردن اندیکسر را نشان می‌دهد. اولین اندیکسر که دارای پارامتری از نوع **int** تحت عنوان **pos** است دقیقاً مشابه **مثال ۱۱-۱** 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 به شکل 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 ۱۱-۲** 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 پیاده سازی آنها آشنا شده اید. با استفاده از اندیکسرها می توان به عناصر یک کلاس همانند یک آرایه دسترسی پیدا کرد. در این مبحث اندیکسهای سرریز شده و چند پارامتری نیز مورد بررسی قرار گرفتند. در آینده و در مباحث پیشرفته تر با موارد بیشتری از استفاده اندیکسرها آشنا خواهید شد.

نکات :

۱. منظور از اندیکسر سرریز شده چیست؟

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

۲. از اندیکسر چگونه مانند آرایه استفاده می شود؟

همانطور که در این درس مشاهده کردید دسترسی به عناصر اندیکسر همانند آرایه ها با استفاده از یک اندیس صورت می پذیرد. با استفاده از این اندیس می توان به عنصر مورد نظر کلاس دسترسی پیدا نمود.

۳. 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 رشته ای نمایش داده می شوند و کاربر با انتخاب یکی از این گزینه ها با برنامه ارتباط برقرار می کند. در حقیقت این عنصر کنترلی یکی از روشهای دریافت اطلاعات از کاربر است با این تفاوت که در این روش ورودی هایی که کاربر می تواند وارد نماید محدود شده هستند و از قبل تعیین شده اند. نمونه ای از یک **ListBox** قسمت انتخاب نوع فونت در برنامه **Word** است که در آن لیستی از فونتهای موجود در سیستم نمایش داده می شود و کاربر با انتخاب یکی از آنها به برنامه اعلام می کند 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 می خواهد تا در هنگام انتخاب یکی از گزینه هایش بتواند اطلاعات خود را بطور خودکار **update** نماید و یا به عبارتی بتواند ورودی دریافت نماید. تمامی این اهداف با استفاده از اندیکسر میسر می شود. اندیکسرها شبیه به **property** ها اعلان می شوند با این تفاوت مهم که **اندیکسرها بدون نام** هستند و نام آنها تنها کلمه کلیدی **this** است و همین **this** مورد اندیکس شدن قرار می گیرد و سایر موارد بشکل پارامتر به اندیکسر داده می شوند.

```
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 می توان با مفهوم آن آشنا شد. برای مثال دسترسی به **ListBox** بشکل زیر است :

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

نمونه برنامه ای 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
```

```
{
```

```
    // رشته های مورد نظر به ListBox تخصیص داده می شوند.
```

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

```
    {
```

```
        // فضایی را برای ذخیره سازی رشته های تخصیص می دهد.
```

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

```
        // رشته های وارد شده به سازنده را درون آرایه ای کپی می کند.
```

```
        foreach (string s in initialStrings)
```

```
        {
```

```
            strings[ctr++] = s;
```

```
        }
```

```
    } //end of constructor
```

```
    // رشته ای به انتهای کنترل افزوده می شود.
```

```
    public void Add(string theString)
```

```
    {
```

```
        if (ctr >= strings.Length)
```

```
        {
```

```
            // در این قسمت می توان 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 می توان 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 بپردازیم. 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 کلاس ها 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) می توان اشیایی با انواع جدید ایجاد کرد که این اشیاء می توانند شبیه به انواع موجود (int, float, ...) باشند. حال سوال اینست که چه زمانی از ساختارها (struct) بجای کلاس استفاده می کنیم؟ 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 شی ای دارید که همانند این انواع رفتار نمایند لازم است تا از ساختارها (struct) استفاده نمایید. در ادامه این مبحث نکات و قوانینی را ذکر می کنیم که با استفاده از آنها بهتر بتوانید از ساختارها (struct) استفاده نمایید.

اعلان و پیاده سازی struct

برای اعلان یک struct کافیت تا با استفاده از کلمه کلیدی struct که بدنبال 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 ۱-۱۲: نمونه ای 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>
/// مثالی از اعلان و ساخت یک struct
/// </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 ۱-۱۲ 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 ۱-۱۲ و **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 ۱-۱۲ 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 پیاده سازی است. هر کلاس و یا ساختاری که از یک **interface** ارث بری نماید باید تمامی متدهای آنرا پیاده سازی 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 پیاده سازی کنید، باید به این نکته توجه کنید که این نوع چگونه مورد استفاده قرار می گیرد. اگر می خواهید سازنده ای بدون پارامتر داشته باشید، در اینصورت کلاس تنها گزینه شماست. همچنین توجه نمایید از آنجائیکه یک ساختار بعنوان یک نوع مقداری در نظر گرفته می شود، در پشته (**Stack**) ذخیره می شود و حال آنکه کلاس در **heap** ذخیره می گردد.

نکات مهم و مطالب کمکی

۱. تفاوت های اصلی بین کلاس و ساختار در چیست؟

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

همانطور که قبلاً نیز اشاره شد شما نمی توانید برای یک **struct** سازنده ای 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 می دهد { ... } 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 نمی کرد. در حقیقت تفاوت در اینست که در مورد **struct**، کامپایلر اجازه ایجاد سازنده پیش فرض جدیدی را به شما نمی دهد ولی در مورد کلاس چنین نیست. هنگام اعلان کلاس در صورتیکه شما سازنده پیش فرضی اعلان نکرده باشید، کامپایلر سازنده ای پیش فرض برای آن در نظر می گیرد ولی در مورد **struct** تنها سازنده پیش فرضی معتبر است که کامپایلر آنرا ایجاد نماید نه شما !

یکی دیگر از تفاوت های 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 نمی داد ولی چون بصورت **struct** تعریف شده خطای زمان کامپایل رخ خواهد داد :

```
struct Time
{
    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
    } // seconds not initialized : خطای زمان کامپایلی بدین صورت رخ می دهد
    □
    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.

۲. 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 تا 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 localVar;
        □
    }
    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 - واسطه‌ها (Interfaces)

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

- ۱- آشنایی با مفهوم کلی واسطه‌ها
- ۲- تعریف یک واسط
- ۳- استفاده از یک interface
- ۴- پیاده‌سازی ارث‌بری در interface ها
- ۵- نکات مهم و پیشرفته
- ۶- مثالی کاربردی از واسطه‌ها
- ۷- منابع مورد استفاده

واسطه‌ها 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 واسطه‌ها 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 بگوئیم 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 همنام 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. (تأثیرات ناخواسته و غیرمنتظره و یا به عبارتی پیش‌بینی 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 تأثیری بر رفتار کاربر نخواهد داشت و حتی کاربر از آن مطلع نیز نمی‌شود. در مفاهیم کلی شیء‌گرایی، **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 تأثیر قرار نمی‌دهد.

از دیدگاه تکنیکی، واسطه‌ها بسط مفهومی هستند که از آن به عنوان انتزاع (**Abstract**) یاد می‌کنیم. در کلاسهای انتزاعی (که با کلمه کلید **abstract** مشخص می‌شدند) سازنده کلاس قدر بود تا فرم کلاس خود را مشخص نماید: نام متدها، نوع بازگشتی

آنها و تعداد و نوع پارامتر آنها، اما بدون پیاده سازی بدنه متد. یک **interface** همچنین می تواند دارای فیلدهایی باشد که تمامی آنها **static** و **final** هستند. یک **interface** تنها یک فرم کلی را بدون پیاده سازی به نمایش می گذارد.

از این دیدگاه، یک واسط بیان می دارد که : " این فرم کلی است که تمامی کلاسهایی که این واسط را پیاده سازی می کنند، باید آنرا داشته باشند." از سوی دیگر کلاسهها و اشیاء دیگری که از کلاسی که از یک واسط مشتق شده استفاده می کنند، می دانند که این کلاس حتماً تمامی متدها و اعضای واسط را پیاده سازی می کند و می توانند به راحتی از آن متدها و اعضا استفاده نمایند. پس به طور کلی می توانیم بگوئیم که واسطها بمنظور ایجاد یک پروتکل (**protocol**) بین کلاسهها مورد استفاده قرار می گیرند. (همچنان که برخی از زبانهای برنامه سازی بجای استفاده از کلمه کلیدی **interface** از **protocol** استفاده می نمایند).

به دلیل اینکه کلاسهها و ساختارهایی که از **interface** ها ارث بری می کنند موظف به پیاده سازی و تعریف آنها هستند، قانون و قاعده ای در این باره ایجاد می گردد. برای مثال اگر کلاس **A** از واسط **IDisposable** ارث بری کند، این ضمانت بوجود می آید که کلاس **A** دارای متد **Dispose()** است، که تنها عضو **interface** نیز می باشد. هر کدی که می خواهد از کلاس **A** استفاده کند، ابتدا چک می نماید که آیا کلاس **A** واسط **IDisposable** را پیاده سازی نموده یا خیر. اگر پاسخ مثبت باشد آنگاه کد متوجه می شود که می تواند 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 می توان هر متدی با انواع مختلف پارامترها و نوع بازگشتی باشد. توجه نمایید همانطور که گفته شد این متد دارای پیاده سازی 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 می باشد. علت این امر آنست که **interface** تنها نوع بازگشتی و پارامترهای متد را مشخص می نماید و کلاس یا شی ای که از آن ارث می برد باید آنرا پیاده سازی نماید. مثال زیر نحوه استفاده از این واسط را نشان می دهد.

مثال ۱-۱۳ : استفاده از واسطها و ارث بری 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 ارث بری از یک کلاس، از واسط **IMyInterface** ارث بری کرده است. حال که این کلاس از واسط مورد نظر ارث بری کرده است، باید، توجه نمایید باید، تمامی اعضای آنرا پیاده سازی کند. در این مثال این عمل با پیاده سازی تنها عضو واسط یعنی متد **MethodToImplement()** انجام گرفته است. توجه نمایید که

پیاده سازی 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 می شود. مثال زیر نحوه ارث بری واسطها از یکدیگر نیز نمایش داده شده است.

مثال ۲-۱۳: ارث بری 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 ۲-۱۳ دارای ۲ واسط است: یکی **IMyInterface** و واسطی که از آن ارث می برد یعنی **IParentInterface**. هنگامیکه واسطی از واسط دیگری ارث بری می کند، کلاس یا ساختاری که این واسطها را پیاده سازی می کند، باید تمامی اعضای واسطهای موجود در سلسله مراتب ارث بری را پیاده سازی نماید. در مثال ۲-۱۳، چون کلاس **InterfaceImplementer** از واسط **IMyInterface** ارث بری نموده، پس از واسط **IParentInterface** نیز ارث بری دارد، از اینرو باید کلیه اعضای این دو واسط را پیاده سازی نماید.

چند نکته مهم:

- ۱- با استفاده از کلمه کلید **interface** در حقیقت یک نوع مرجعی (**Reference Type**) جدید ایجاد نموده اید.
- ۲- از لحاظ نوع ارتباطی که واسطها و کلاسها در ارث بری ایجاد می نمایند باید به این نکته اشاره کرد که، ارث بری از کلاس رابطه "است" یا "بودن" (**is-a relation**) را ایجاد می کند (ماشین یک وسیله نقلیه است) ولی ارث بری از یک واسط یا **interface** نوع خاصی از رابطه، تحت عنوان "پیاده سازی" (**implement relation**) را ایجاد می کند.

"می توان ماشین را با وام بلند مدت خرید" که در این جمله ماشین می تواند خریداری شدن بوسیله وام را پیاده سازی کند).

۳- فرم کلی اعلان **interface** ها بشکل زیر است :

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

که در اعضای آن بشرح زیر می باشند :

attributes : صفتهای واسط

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

interface-name : نام واسط

:base-list : لیست واسطهایی که این واسط آنها را بسط می دهد.

Interface-body : بدنه واسط که در آن اعضای آن مشخص می شوند

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

۴- هدف از ایجاد یک **interface** تعیین تواناییهاییست که می خواهیم در یک کلاس وجود داشته باشند.

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

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

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

interface IStorable

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

حال می خواهید کلاسی با عنوان **Document** ایجاد نمایید که این کلاس باید قابلیت خواندن و نوشتن از/به پایگاه داده را داشته باشد، پس می توانید کلاس را از روی واسط **IStorable** پیاده سازی کنید.

public class Document : IStorable

```
{
public void Read( ) {...}
public void Write(object obj) {...}
// ...
}
```

حال بعنوان طراح برنامه، شما وظیفه داری تا به پیاده سازی این واسط بپردازید، بطوریکه کلیه نیازهای شما را برآورده نماید. نمونه ای از این پیاده سازی در مثال ۳-۱۳ آورده شده است.

مثال ۳-۱۳ : پیاده سازی واسط و ارث بری - مثال کاربردی

using System;

// اعلان interface

```
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
IStorable Status: 0

```


- ۶- 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.
- ۷- توجه نمایید که از روی یک واسط نمی توان نمونه ای جدید ایجاد کرد بلکه باید کلاسی از آن ارث بری نماید.
- ۸- کلاسی که از واسط ارث بری می کند باید تمامی متدهای آنرا دقیقاً 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 - رخدادهای و delegate ها در C#

نکته مهم قبل از مطالعه این درس

توجه نمایید، **delegate** ها و رخدادهای بسیار با یکدیگر در تعامل اند، از اینرو در برخی موارد، قبل از آموزش و بررسی رخدادهای، به ناچار، از آنها نیز استفاده شده و یا به آنها رجوع شده است. رخدادهای 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 رخدادهای 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 نمونه های جدید از یک delegate (بخش پیشرفته)
- درک اینکه یک رخداد یا یک event چیست؟
- اعلان رخدادهای
- نکات و توضیحات پیشرفته
- ثبت شدن در یک رخداد
- لغو عضویت در یک رخداد
- فراخوانی رخدادهای
- مثالی پیشرفته از استفاده رخدادهای در فرمهای ویندوز
- نکات کلیدی درباره رخدادهای و delegate ها
- منابع مورد استفاده

طی درسهای گذشته، چگونگی ایجاد و پیاده سازی انواع مرجعی (**Reference Type**) را با استفاده از ساختارهای زبان **C#**، یعنی **کلاسها (Class)** و **واسطهها (Interface)**، فرا گرفتید. همچنین فرا گرفتید که با استفاده از این انواع مرجعی، میتوانید نمونه های جدیدی از اشیاء را ایجاد کرده و نیازهای توسعه نرم افزار 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 واسطهها، یکسری از صفات و رفتارها را تعریف می کردیم تا فرم کلی داشته باشیم و تمام اشیاء خود به پیاده سازی این صفا و رفتارها می پرداختند. در این درس با یکی دیگر از انواع مرجعی (**Reference Type**) در زبان **C#** آشنا خواهید شد.

مقدمه ای بر رخدادهای و delegate ها

در گذشته، پس از اجرای یک برنامه، برنامه مراحل اجرای خود را مرحله به مرحله اجرا می نمود تا به پایان برسد. در صورتیکه نیاز به ارتباط و تراکنش با کاربر نیز وجود داشت، این امر محدود و بسیار کنترل شده صورت می گرفت 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 گزینه ای خاص در یک منو، کلیک کردن بر روی دکمه ها برای انجام عملیاتی خاص و ... رهیافتی که امروزه در برنامه نویسی مورد استفاده است، تحت عنوان "برنامه نویسی بر پایه رخدادها" (Event-Based Programming) شناخته می شود. در این رهیافت برنامه همواره منتظر انجام عملی از سوی کاربر می ماند و پس از انجام عملی خاص، رخداد مربوط به آن را اجرا می نماید. هر عمل کاربر باعث اجرای رخدادی می شود. در این میان برخی از رخدادها بدون انجام عملی خاص از سوی کاربر اجرا می شوند، 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 می باشند. 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 می شوند.

از delegate، می توان به عنوان یک Callback یاد نمود، بدین معنا که یک کلاس می تواند 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 می توان 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 می توانید 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 برنامه نویسی کرده اید، حتماً با این مفهوم آشنایی دارید. در زبان C این مفهوم با اشاره گرها (pointer) بیان می شود. اما برای افرادی که با زبانهای دیگری برنامه نویسی می کرده اند 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 می توان گفت که delegate نوعی است شبیه به متد و همانند آن نیز رفتار می کند. در حقیقت delegate انتزاعی (Abstraction) از یک متد است. در برنامه نویسی ممکن به شرایطی برخورد کرده باشید که در آنها می خواهید عمل خاصی را انجام دهید اما دقیقاً نمی دانید که باید چه متد یا شیء ای را برای انجام آن عمل خاص مورد استفاده قرار دهید. در برنامه های تحت ویندوز این گونه مسائل مشهودتر هستند. برای مثال تصور کنید در برنامه شما، دکمه ای قرار دارد که پس از فشار دادن این دکمه توسط کاربر شیء ای یا متدی باید فراخوانی شود تا عمل مورد نظر شما بر روی آن انجام گیرد. می توان 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 الگوریتم هایی طراحی می نائیم که این الگوریتمهای کارهای خاصی را با استفاده از متدها انجام می دهد، ابتدا متغیرهایی مقدار دهی شده و سپس متدی جهت پردازش آنها فراخوانی می گردد. حال در نظر بگیرید که به الگوریتمی نیاز دارید که بسیار قابل انعطاف و قابل استفاده مجدد (reusable) باشد و همچنین در شرایط مختلف قابلیت های مورد نظر را در اختیار شما قرار دهد. تصور کنید، به الگوریتمی نیاز دارید که از نوعی از ساختمان داده پشتیبانی کند و همچنین می خواهید این ساختمان داده را در مواردی مرتب (sort) نمایید، بعلاوه می خواهید 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** و 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 بحث 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 ۱-۱۴ 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 ۱-۱۴ : 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 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** بسیار شبیه به اعلان متد است، با این تفاوت که دارای کلمه کلیدی **delegate** در اعلان است و در انتهای اعلان آن ";" قرار می گیرد و نیز پیاده سازی ندارد. در زیر اعلان **delegate** که در مثال ۱-۱۴ آورده شده را مشاهده می نمایید :

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

این اعلان، مدل متدی را که **delegate** می تواند به آن اشاره کند را تعریف می نماید. متدی که می توان از آن بعنوان **delegate handler** برای **Comparer** استفاده نمود، هر متدی می تواند 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 ۱-۱۴ مورد استفاده قرار گرفته است، نشان داده شده است :

```
public static int ComparerFirstNames(object name1, object name2)
```

```
{
    ...
}
```

برای استفاده از **delegate** می بایست نمونه ای از آن ایجاد کنید. ایجاد نمونه جدید از **delegate** همانند ایجاد نمونه ای جدید از یک کلاس است که به همراه پارامتری جهت تعیین متد **delegate handler** ایجاد می شود :

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

در مثال ۱-۱۴، **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 می توان **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 می پردازیم. در اینجا این مثال را یکبار بدون استفاده از **delegate** و بار دیگر با استفاده از آن حل کرده و بررسی می نمائیم. 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 تعامل اند، از اینرو در برخی موارد به ناچار از رخدادها نیز استفاده شده است. رخدادها در قسمت انتهایی این درس آورده شده اند، 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.

حل 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 پیاده سازی 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 (بعنوان مثال در برنامه‌های مبتنی بر **GUI** در اینگونه برنامه‌ها حجم عظیمی از رخدادها 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 نمی کرد :

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

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

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

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

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

فراخوانی delegate

قدرت و توانایی **delegate** زمانی مشهود می گردد که می خواهید از آن استفاده نمایید. برای مثال، با متغیر **param** در مثال قبل چکار می توانید انجام دهید؟ اگر **param** متغیری از نوع **int** بود، از مقدار آن استفاده می کردید و با استفاده از عملگرهایی نظیر **+**، **-** و یا عملگرهای مقایسه ای، عملی خاص را بر روی آن انجام می دادید. اما حال که **param** متغیری از نوع **int** نیست، چه می کنید؟ متغیر **param** یک **delegate** است و همانطور که گفته شد، **delegate** انتزاعی از یک متد است، پس هر عملی که متد انجام می دهد، **delegate** نیز می تواند انجام دهد. با استفاده از پرانتز، می توان از **delegate** استفاده نمود :

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

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

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 نمی توان به آن ارسال نمود و نمی توان 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 می دهد
    int hhg = param();   // خطای زمان کامپایل رخ می دهد
    Console.WriteLine(param()); // خطای زمان کامپایل رخ می دهد
}
```

توجه نمایید که **delegate** را به هر نحوی می توانید 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 seconds)
    {
    }
}
```

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 **Ticker** 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 می دهیم تا متدی را که بهنگام روی دادن رخداد فراخوانی می شود، معین گردد. مثال زیر روش تعریف رخداد را نشان می دهد.

مثال ۲-۱۴: اعلان و پیاده سازی 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 می شود. 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 می کنید، **delegate** ها تنها با استفاده از **+=** می توانند به رخدادها افزوده شوند. در این مثال هر گاه **MyEvent** فراخوانی شود، متد **CallbackMethod** اجرا می شود 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 می توان نوشت. این نسخه از مثال ۲-۱۴ 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 **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 seconds)
    {
        □
    }
    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 پراانتز 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 می شود 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 می شود، 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** می توانند **tick** را فرا بخوانند.

```

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

```

مثالی پیشرفته از استفاده رخدادها در فرمهای ویندوز

حال که تا حدودی با رخدادها و ساختار آنها آشنا شدید، در این قسمت قصد دارم تا مقداری درباره استفاده رخدادها در فرمهای ویندوز و **GUI** ها صحبت نمایم. هر چند تا کنون کلیه برنامه ها و مطالبی که مشاهده کرده اید مبتنی بر **Console** بوده اند، اما به علت استفاده بیشمار رخدادها در فرمهای ویندوز و برنامه های 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 رئوس 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 پیشرفته تر رخدادها در برنامه نویسی بوده است.

کلاسهای **GUI** مربوط به **.Net Framework**. بطور گسترده ای از رخدادها استفاده می نمایند. در مثالی که در اینجا مورد بررسی قرار می دهیم، برنامه ای 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 می شوند. (**Button** عضو **System.Windows.Forms** است). کلاس **Button** از کلاس **Control** ارث بری می کند 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 پیاده سازی آنرا نشان می دهد. همانطور که ملاحظه می نمایید، 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 ارث بری کلاس **Button** از کلاس **Control** نیز مشخص شود.

پس از اینکه بر روی دکمه ای واقع در فرم ویندوز کلیک کنید، **Button** بطور خودکار رخداد **Click** را فرا می خواند. هدایت این پروسه باعث می شود تا بتوان به سادگی **delegate** ی برای کنترل این رخداد ایجاد نمود. در مثالی که در زیر مشاهده می کنید، دکمه ای 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.okay_Click);
    }

    private void okay_Click(object sender, System.EventArgs args)
    {
    }
}

```

همانطور که مشاهده می کنید، کلاس **Example** از **System.Windows.Forms.Form** مشتق می شود، از اینرو تمامی خواص آن را به ارث می برد. **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 می شود.

این کد، شبیه به کدهایی است که توسط محیط های برنامه سازی نظیر **Visual Studio.Net** و یا **C#Builder** بطور خودکار تولید می شوند و تنها کافیست 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 رخدادها 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.

مطالب این درس در اینجا به پایان رسید. امیدوارم مورد قبول شما بوده باشد. خواهشمندم نظرات و پیشنهادات ارزنده خود را برای من ایمیل کنید تا از آنها در جهت بهبود مطالب سایت استفاده گردد.

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 :

- (۱) درک و فهم صحیح یک استثناء یا Exception
- (۲) پیاده‌سازی یک روتین برای برخورد با استثناها بوسیله بلوک try/catch
- (۳) 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 متد ها، 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 استثنائایی، 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 نمائیم، احتمال وقوع (Thrown) یکی از استثناهای زیر وجود دارد :

SecurityException
ArgumentException
ArgumentNullException
PathTooLongException
DirectoryNotFoundException
UnauthorizedAccessException
FileNotFoundException
NotSupportedException

با نگاهی بر مستندات Net Framework SDK، به سادگی می‌توان از خطاها و استثنائایی که ممکن است یک متد ایجاد کند، مطلع شد. تنها کافیست به قسمت Reference/Class Library رفته و مستندات مربوط به Namespace/Class/Method را مطالعه نمائید. 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 بپردازید.

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 ۱-۱۵ 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 ۱-۱۵ 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 می شود.

نکته : توجه نمایید که کلیه مثالهای موجود در این درس به طور تعمیدی دارای خطاهایی هستند تا شما با نحوه مقابله با استثناءها 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 ۱-۱۵ تنها دارای یک بلوک `catch` است، اما تمامی استثنایایی که ممکن است رخ دهند را نشان داده و مورد بررسی قرار می دهد زیرا از نوع کلاس پایه استثناء، یعنی `Exception` تعریف شده است. در کنترل و مقابله با استثناءها، باید استثنای خاص را زودتر از استثنای کلی مورد بررسی قرار داد. کد زیر نحوه استفاده از چند بلوک `catch` را نشان می دهد :

```
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 می گیرد. 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 می توان بدست آورد.

استثنایایی که مورد بررسی قرار نگیرند، در بالای `Stack` نگهداری می شوند تا زمانیکه 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 می گردد. این چنین حالتی بسیار نا مناسب 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 می شود.

یکی از حالتیهای 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 استثناءها 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 ۲-۱۵، 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 ۲-۱۵، `outStream` با موفقیت باز می شود، بدین معنا که برنامه `handle` ای به یک فایل باز شده در اختیار دارد. اما زمانی که 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 outStream = null;
        FileStream inStream = null;
        try
        {
            outStream = File.OpenWrite("DestinationFile.txt");
            inStream = File.OpenRead("BogusInputFile.txt");
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
        finally
        {
            if (outStream != null)
            {
                outStream.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 استنهاها بوسیله بلوکهای `try/catch` پیاده سازی نمایید. علاوه می توانید با استفاده از بلوک `finally` مطمئن باشید که 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 :

- ۱- صفتها چه هستند و چرا از آنها استفاده می کنیم
- ۲- استفاده از صفتهای تک پارامتری و چند پارامتری
- ۳- انواع پارامترهای صفت (پارامترهای Named و Positional)
- ۴- Target های صفتها (عناصری که صفتها بر روی آنها اعمال می شوند)
- ۵- تولید صفتهای شخصی
- ۶- تعریف و یا کنترل موارد استفاده از یک صفت
- ۷- استفاده از پارامترهای Named و Positional در صفتهای شخصی
- ۸- انواع (type) معتبر برای پارامترهای صفت
- ۹- استفاده از صفتها در زمان اجرا
- ۱۰- خلاصه مطالب
- ۱۱- منابع

صفتها در حقیقت اطلاعات توضیحی هستند که می توانید آنها را به برنامه های خود بیفزایید. صفتها را می توان برای کلیه عناصر برنامه از قبیل کلاسها، واسطها، اسمبلی ها و ... مورد استفاده قرار داد. از این اطلاعات می توان برای موارد متنوعی در زمان اجرای برنامه استفاده نمود. برای مثال می توان به صفتی مانند **DllImportAttribute** اشاره کرد که امکان برقراری ارتباط با توابع کتابخانه ای Win32 را فراهم می نماید. همچنین صفتهایی نیز وجود دارند که برنامه نویسی یا توسعه دهنده برنامه را در امر تولید برنامه یاری می نمایند. برای مثال می توان به صفت **ObsoleteAttribute** اشاره کرد که با استفاده از آن، در زمان کامپایل برنامه پیغامی برای برنامه نویس نمایش داده می شود و مشخص می کند که متدی خاص مورد استفاده قرار نگرفته و یا دیگر مورد استفاده نیست. همچنین هنگامیکه با فرمهای ویندوز کار می کنیم، صفتهای بسیاری وجود دارند که امکان استفاده از این فرمها را فراهم کرده و باعث می شوند تا اطلاعات مربوط به این عناصر در **property** فرم ظاهر شوند. یکی دیگر از موارد استفاده از صفتها در مسایل امنیتی اسمبلی های **.Net** است. برای مثال صفتهایی وجود دارند که باعث جلوگیری از فراخوانی های غیر مجاز می شوند، بدین معنی که تنها اجازه فراخوانی را به متدها یا اشیایی می دهند که قبلا تعریف شده و مشخص شده باشند.

یکی از علت های استفاده از صفتها آنست که، اغلب سرویس هایی را که آنها برای کاربر فراهم می نمایند، بسیار پیچیده است و با کدهای معمولی نمی توان آنها را بدست آورد. از اینرو استفاده از صفتها در بسیاری از موارد ضروری و اجتناب ناپذیر است. همانطور که خواهید دید، صفتها به برنامه های ما **Metadata** اضافه می نمایند. پس از کامپایل برنامه های **C#**، فایل اسمبلی برای آن ایجاد می گردد که این اسمبلی معمولا یا یک فایل اجرایی است و یا یک **Dll** است. توصیف اسمبلی، در **Metadata** ی مربوط به آن قرار می گیرد. طی پروسه ای تحت عنوان **Reflection**، صفت یک برنامه از طریق فایل **Metadata** ی موجود در اسمبلی آن قابل دسترسی می گردد. (برای آشنایی بیشتر با اسمبلی و **Metadata** می توانید به "کامپایل یک برنامه سی شارپ" در همین سایت مراجعه نمایید.) در حقیقت صفتها، کلاس هایی هستند که می توانید 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 ۱۶-۱](#) موارد متنوعی از استفاده صفت **ObsoleteAttribute** را مشاهده می‌نمایید.

مثال ۱۶-۱ : 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 ۱۶-۱](#) 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 می افزاید که آن را با دو اعلان قبلی متمایز می نماید. نتیجه هر سه اعلان این صفت در زیر آورده شده است. این پیغامها، پیامهای کامپایلر **C#** هستند که به هنگام کامپایل برنامه تولید شده اند.

```
>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 می کنید، 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 می شود که تنها پیغامی ساده تولید گردیده است.

مثال ۱-۱۶ شامل صفت دیگری نیز می باشد. این صفت **STAThreadAttribute** است که معمولاً در ابتدای کلیه برنامه های **C#** و قبل از آغاز متد **Main()** قرار می گیرد. این صفت بیان می دارد که برنامه **C#** مورد نظر می تواند با کد مدیریت نشده **COM** از طریق **Simple Threading Apartment** ارتباط برقرار نماید. استفاده از این صفت در هر برنامه ای می تواند مفید باشد، چراکه شما بعنوان برنامه نویس هیچ گاه اطلاع ندارید که آیا کتابخانه ثالثی که از آن استفاده می کنید، 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 می توانند دارای چندین پارامتر باشند. در **مثال ۲-۱۶**، استفاده از دو پارامتر برای یک صفت نشان داده شده است.

مثال ۲-۱۶

```
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 ۲-۱۶** مشاهده می کنید، صفت مورد استفاده دارای دو پارامتر است. پارامتر اول که یک جمله متنی است و همانند **مثال ۱-۱۶** 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 ۱-۱۶ لحاظ نکردیم. همانطور که در مثال ۲-۱۶ 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 می گیرند و همیشه اختیاری هستند. در [مثال ۳-۱۶](#) صفت **DllImport** را مشاهده می نمایید که دارای هر دو نوع پارامتر **positional** و **named** است.

مثال ۳-۱۶

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

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

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

صفت **DllImport** در [مثال ۳-۱۶](#) دارای یک پارامتر **positional** ("**User32.dll**") و یک پارامتر **named** (**EntryPoint="MessageBox"**) است. پارامترهای **named** در هر مکانی می توانند قرار گیرند و مانند پارامترهای **positional** دارای محدودیت مکانی نیستند. بدین معنا که چون در پارامترهای **named**، نام پارامتر مستقیماً مورد استفاده قرار می گیرد، محل قرار گیری آن در لیست پارامترهای صفت مهم نیست اما در مورد پارامترهای **positional** چون اسم پارامتر مورد استفاده قرار نمی گیرد، 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 می توان 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 نمی باشد.

Target های صفتها (عناصری که صفتها بر روی آنها اعمال می شوند)

صفتهایی که تا کنون مشاهده کردید، همگی بر روی متدها اعمال شده بودند. اما عناصر مختلف دیگری در **C#** وجود دارند که می توان صفتها را بر روی آنها اعمال نمود. جدول ۱-۱۶ عناصر مختلف زبان **C#** را که صفتها بر روی آنها اعمال می شوند را نشان می دهد.

عناصر اعمال شونده	قابل اعمال به
all	به تمامی عناصر قابل اعمال هستند.
assembly	به تمام یک اسمبلی
class	کلاسها
constructor	سازنده ها
delegates	Delegate ها
enum	عناصر شمارشی
event	رخدادها
field	فیلدها
interface	واسطها
method	متدها
module	ماژولها (کدهای کامپایل شده ای که می توانند 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 می توان 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 می شود و باعث ارتباط با **CLS** می گردد، صفت **CLSCompliantAttribute** است. **CLS** یا همان **Common Language Specification** امکان برقراری ارتباط بین کلیه زبانهایی که تحت **.Net** کار می کنند را فراهم می نماید. **Target** های صفتها با استفاده از اسم **Target** که بعد از آن کولون قرار می گیرد، ایجاد می شوند. در مثال ۴-۱۶ نحوه استفاده از این صفت نشان داده شده است.

مثال ۴-۱۶

```
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 ۴-۱۶ 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 ۴-۱۶ 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 صفتهای 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 ۵-۱۶ توجه فرمایید.

مثال ۵-۱۶

```

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 ۶-۱۶

```
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
{
}
```

همانطور که مشاهده می‌کنید 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 می‌شود.

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 می‌توانیم موارد استفاده از صفتی را که تولید کرده‌ایم را کنترل کنیم.

این کلاس دارای سه **property** مختلف است که می‌توان آنها را به هنگام استفاده صفت شخصی تنظیم نمود و مورد استفاده قرار داد.

ValidOn

با استفاده از این **property** می‌توانیم مشخص کنیم که صفت تولید شده توسط ما، بر روی کدام یک از عناصر برنامه قابل اعمال هستند. اطلاعات این عناصر از **AttributeTarget** گرفته می‌شود و می‌توان عناصر مختلف را بوسیله **OR** بیتی با یکدیگر ترکیب نمود.

AllowMultiple

با استفاده از این **property** می‌توان مشخص کرد که آیا می‌توان 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 ۷-۱۶ را بررسی نمایید.

مثال ۷-۱۶

```
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 می کند 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 می دهیم 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 ۸-۱۶

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false,
Inherited = false)]
public class HelpAttribute : Attribute
{
    public HelpAttribute(String Description_in)
    {
        this.description = Description_in;
        this.verion = "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 **Metadate** مربوط به یک اسمبلی، لیست کاملی از کلاسها، انواع و متدهایی را که برای آن اسمبلی خاص تعریف شده‌اند را، بدست آوریم. به مثال ۹-۱۶ در این باره توجه نمایید.

مثال ۹-۱۶: 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 می توان ساعتها در مورد صفتهای مختلف بحث نمود. اما آنچه مسلم است تمرین مستمر و پیگیری برنامه نویسی در یافتن مواردی که می توان با استفاده از صفتها، برنامه ای 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 می شوند 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.