

آموزش زبان C#

۱. مفاهیم اولیه زبان C# : فهم ساختار پایه‌ای یک برنامه در C#، آشنایی با Namespace، آشنایی با کلاس و
۲. متغیرها، عبارات و عملگرها : آشنایی با متغیرها، آشنایی با انواع پیش فرض زبان، آشنایی با عبارات، آشنایی با رشته‌ها، آشنایی با آرایه‌ها و
۳. دستورالعمل‌های کنترلی - شرطی : آشنایی با دستور if، آشنایی با دستور switch و break، آشنایی با دستور goto و
۴. دستورالعمل‌های کنترلی - حلقه‌ها : آشنایی کلی با حلقه‌ها - حلقه while - حلقه do while - حلقه for و foreach و
۵. متدها : ساختار یک متد، انواع متدها، ایجاد نمونه جدید از اشیاء، انواع پارامترهای متد، فراخوانی متد، آشنایی با کلمه کلیدی this و
۶. Namespaces : آشنایی با Namespace - هدایتگر using - هدایتگر alias - اعضای یک Namespace و
۷. کلاسها : درک کاملتر کلاسها، سازنده و تخریب کننده در کلاسها، اعضای کلاس و

۸. ارث‌بری کلاسها : پیاده‌سازی کلاس پایه ، کلاسهای مشتق شده، مقدار دهی کلاس پایه، فراخوانی اعضای کلاس پایه، پنهان سازی اعضای کلاس
۹. چند ریختی Polymorphism : آشنایی با مفهوم چند ریختی، متدهای مجازی، Override کردن متدها، استفاده از چندریختی در برنامه‌ها و
۱۰. ویژگیها (Properties) : موارد استفاده، پیاده‌سازی و ایجاد property، ایجاد property های فقط خواندنی و فقط نوشتنی و
۱۱. اندیکسر (Indexer) : پیاده‌سازی اندیکسر، سرریزی اندیکسر، اندیکسهای چند پارامتری، نکات پیشرفته درباره اندیکسرها و
۱۲. ساختارها (Structures) : ساختار چیست؟، پیاده‌سازی و استفاده از یک ساختار، تفاوت ساختار و کلاس و
۱۳. واسطها و رابطها (Interfaces) : مفهوم کلی واسطها، تعریف یک واسط، استفاده از interface، پیاده‌سازی ارث‌بری در واسطها و
۱۴. رخداد (Event) و delegate : درک مفهوم delegate، رخداد چیست؟ پیاده‌سازی رخدادها و delegate ها و
۱۵. Handling Exception : درک مفهوم استثناء و چگونگی برخورد با آن، بررسی بلوکهای try/catch برای مقابله با استثناء و بلوک finally
۱۶. صفتها (Attribute) : صفتها چه هستند و چگونه از آنها استفاده می‌کنیم، پارامترهای صفتها و انواع آنها، ساخت صفتهای شخصی و
۱۷. انواع شمارشی (Enum) : انواع شمارشی چه هستند، چگونه تولید شده و چگونه مورد استفاده قرار می‌گیرند، موارد استفاده از آنها و
۱۸. Overload کردن عملگرها : در این بحث با نحوه Overload کردن عملگرها آشنا خواهید شد.
۱۹. ساختمان داده‌ها در سی شارپ (۱) : در این بحث با مفاهیم اولیه ساختمان داده آشنا می‌شوید. همچنین به معرفی ساختمان داده‌های مهم و رایج نظیر آرایه و لیست نیز پرداخته شده است.

درس اول: آغاز کار با C#

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

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

لیست 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 برنامه‌سازی دو نوع کامپایلر وجود دارد. یک نوع کامپایلر که به کامپایلر Command Line معروف است و نوع دیگر کامپایلر Visual است. کامپایلرهای Command Line محیطی شبیه به محیط

DOS دارند و با دادن یک سری دستورات به اجرا در می‌آیند. کامپایلرهای *Visual* محیطی همانند ویندوز دارند که با دارا بودن محیط گرافیکی و ابزارهای خاص، برنامه‌نویس را در امر برنامه‌سازی کمک می‌کنند. از نمونه‌های هر یک از کامپایلرها، می‌توان به *Microsoft C#* و *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 می‌شود.

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

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

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

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

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

```
Console.ReadLine();
```

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

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

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

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

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

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

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

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

در زبان C#، برای قرار دادن توضیحات در کد برنامه از // استفاده می‌کنیم. بدین معنا که کامپایلر در هنگام اجرای برنامه توجهی به این توضیحات نمی‌کند و این توضیحات تنها به منظور بالا بردن خوانایی متن و جهت و کمک به فهم بهتر برنامه قرار می‌گیرند. استفاده از // تنها در مواردی کاربرد دارد که توضیحات شما بیش از یک خط نباشد. در صورت تمایل برای استفاده از توضیحات چند خطی باید در ابتدای شروع

توضیحات از `/*` و در انتها آن از `*/` استفاده نمایید. در این حالت تمامی مطالبی که بین `/*` `*/` قرار می‌گیرند به عنوان توضیحات (Comments) در نظر گرفته می‌شوند.

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

بسیاری از برنامه‌ها از کاربر ورودی دریافت می‌کنند. انواع گوناگونی از این ورودی‌ها می‌توانند به عنوان پارامتری برای متد `Main()` در نظر گرفته شوند. لیست ۱-۲ برنامه‌ای را نشان می‌دهد نام کاربر را از ورودی دریافت کرده و آن را بر روی صفحه نمایش می‌دهد. این ورودی به صورت پارامتری برای متد `Main()` در نظر گرفته شده است.

لیست ۱-۲ : برنامه‌ای که ورودی را از کاربر، بعنوان پارامتر `Main()`، دریافت می‌کند.

`// Namespace` اعلان

`using System;`

`//` کلاس آغازین برنامه

`class` NamedWelcome

`{`

`//` آغاز اجرا برنامه

`public static void` Main(`string[]` args)

`{`

`//` نمایش بر روی صفحه

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

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

`}`

`}`

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

همان طور که در لیست ۱-۲ مشاهده می‌نمایید، پارامتر متد `Main()` با عنوان `args` مشخص شده است. با استفاده از این نام در داخل متد می‌توان آن استفاده نمود. نوع این پارامتر از نوع آرایه‌ای از نوع رشته (`string[]`) در نظر گرفته شده است. انواع (`types`) و آرایه‌ها را در درس‌های بعدی بررسی می‌کنیم. فعلاً بدانید که آرایه رشته‌ای جهت نگهداری چندین کاراکتر مورد استفاده قرار می‌گیرد. [] مشخص کننده آرایه هستند که مانند یک لیست عمل می‌کند.

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

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

```
>Hello!, Meysam!
```

```
>Welcome to C# Persian Tutorial!
```

همان گونه که می‌بینید، پس از اجرای برنامه نام شما که از طریق `Command-Line` آنرا وارد نموده‌اید در خروجی ظاهر می‌شود. 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 می‌آید. مقدار `n` از صفر آغاز شده و به ترتیب افزایش می‌یابد. 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 می‌دهد. ملاحظه می‌نمایید که چون ۳ مقدار را می‌خواهیم نمایش دهیم، سه بار از آرگومان `{n}` استفاده کرده‌ایم و هر بار یک واحد به مقدار قبلی افزوده‌ایم. در آینده بیشتر با این مفاهیم آشنا می‌شویم.

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 نمی‌شود. 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 ۲-۱ 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 بازمی گرداند. این مقدار در این برنامه به آرگومان `{0}` برگردانده می شود. این خط از برنامه را می توان طور دیگری هم نوشت :

```
string myName=Console.ReadLine();
```

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

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

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

در اینجا اولین درس به پایان رسید. از همه دوستانی که این مطلب را می خوانند تقاضا دارم نظرات و پیشنهادات خود را برای من mail کنند.

درس دوم – عبارات، انواع و متغیرها در C#

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

- آشنایی با متغیرها
- فراگیری انواع (Types) ابتدایی در C#
- فراگیری و درک عبارات (Expressions) در C#
- فراگیری نوع رشته‌ای (String) در زبان C#
- فراگیری چگونگی استفاده از آرایه‌ها (Arrays) در زبان C#

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

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

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

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

It is True that C# Persian provides C# programming language content.

The statement above is not False.

جدول زیر تمامی انواع عددی صحیح C#, اندازه آنها و رنج قابل قبول آنها را نشان می‌دهد.

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

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.

رنج قابل قبول	دقت	اندازه به بیت	نوع
تا	۷ رقم	۳۲	float
تا	۱۵-۱۶ رقم	۶۴	double
تا	۲۸-۲۹ رقم دسیمال	۱۲۸	decimal

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

شرکت پذیری	عملگر(ها)	نوع عمل
------------	-----------	---------

عملیات ابتدایی	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked	از چپ
عملیات یکانی	+ - ! ~ ++x --x (T)x	از چپ
عملیات ضربی	* / %	از چپ
عملیات جمعی	+ -	از چپ
عمل شیفت	<< >>	از چپ
عملیات رابطه‌ای	< > <= >= is	از چپ
عملیات تساوی	= = !=	از راست
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 چپ
عمل انتساب	= *= /= %= += -= <<= >>= &= ^= =	از راست

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

مثال 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 = (~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 می‌گیرند $post-decrement$ و $post-increment$) ابتدا مقدار فعلی عملوند (operand) خود را باز می‌گرداند و سپس به عملوند خود یک واحد اضافه کرده یا از آن یک واحد می‌کاهند. عملگر $++$ یک واحد به عملوند خود اضافه می‌کند و عملگر $--$ یک واحد از عملوند خود می‌کاهد. بدین ترتیب عبارت $x++$ معادل است با عبارت $x=x+1$ یا $x+=1$ اما همانطور که گفته شد باید توجه داشته باشید که این عملگرها ($++$ و $--$) ابتدا مقدار فعلی عملوند خود را برگشت می‌دهند و سپس عمل خود را روی آنها انجام می‌دهند. 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 می‌شود و سپس به متغیر y یک واحد افزوده می‌شود، در صورتیکه اگر این عبارت را بصورت $x=++y$ بنویسیم در اولین اجرای برنامه، ابتدا به مقدار متغیر y یک واحد افزوده می‌شود و سپس این مقدار به متغیر x تخصیص داده می‌شود که در این حالت مقدار متغیر x برابر با ۲ می‌شود. (در مورد عملگر $--$ نیز چنین است.) پس با این توضیح می‌توان گفت که دو عملگر $x++$ و $x--$ ابتدا به عملوند خود یک واحد اضافه یا یک واحد از آن کم می‌کنند و سپس مقدار آنها را باز می‌گردانند.

در مثال ۲، مقدار متغیر $unary$ در قسمت اعلان برابر با ۰ قرار گرفته است. هنگامیکه از عملگر $x++$ استفاده می‌کنیم، به مقدار متغیر $unary$ یک واحد افزوده می‌شود و مقدارش برابر با ۱ می‌گردد و سپس این مقدار، یعنی ۱، به متغیر $preIncrement$ تخصیص داده می‌شود. عملگر $x--$ مقدار متغیر $unary$ را به ۰ باز می‌گرداند و سپس این مقدار را به متغیر $preDecrement$ نسبت می‌دهد.

هنگامیکه از عملگر $x--$ استفاده می‌شود، مقدار متغیر $unary$ ، یا همان مقدار صفر، به متغیر $postDecrement$ تخصیص داده می‌شود و سپس از مقدار متغیر $unary$ یک واحد کم شده و مقدار این متغیر به ۱- تغییر می‌کند. سپس عملگر $x++$ مقدار متغیر $unary$ ، یعنی همان ۱-، را به متغیر $postIncrement$ تخصیص می‌دهد و سپس یک واحد به مقدار متغیر $unary$ می‌افزاید 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 (~) (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 کاربرد ترین 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][];
```

```
        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 نگداری 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 صفره‌جویی حافظه بسیار موثر می‌باشند. یک آرایه دندانه‌دار، در حقیقت یک آرایه تک بعدی است که هر یک از اعضای آن خود یک آرایه تک بعدی می‌باشند. اندازه این عناصر می‌تواند متفاوت باشد و تفاوت این آرایه‌ها با آرایه‌های چند بعدی در همین جا نمایان می‌شود. استفاده از این آرایه‌ها در مواردی کاربرد دارد که نیازی نیست تا تمامی ابعاد آرایه دارای تعداد عناصر مساوی باشند. بعنوان مثال فرض کنید می‌خواهید آرایه‌ای جهت نگهداری تعداد روزهای ماههای مختلف سال تهیه کنید. در صورتیکه بخواهید از آرایه چند بعدی استفاده نمایید، چون تعداد روزهای تمامی ماههای سال یکسان نیست، مجبورید تا تعداد عناصر تمام بعدهای آرایه را برابر با بزرگترین تعداد روز ماهها، یعنی 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 می‌توان به هر یک از ماههای سال تعداد روزهای مورد نظر آن ماه را تخصیص داد و فضایی بلا استفاده ایجاد نخواهیم کرد. توجه نمایید که چون تعداد عناصر بعد دیگر این آرایه معین نشده است در برنامه باید این تعداد عنصر را مشخص نماییم :

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

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

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

دسترسی به عناصر آرایه از طریق اندیس امکان پذیر است. اندیس شماره محل ذخیره سازی داده های ما می باشد که با دادن این شماره می توانیم به داده مورد نظر دسترسی پیدا کنیم. در C# همانند C و C++ اندیس خانه های آرایه از صفر آغاز می گردد یعنی اولین خانه آرایه دارای شماره صفر است و عناصر بعدی به ترتیب یک واحد به اندیسشان اضافه می گردد. پس شماره اندیس آرایه همیشه یک واحد کمتر از تعداد عناصر آن است، یعنی آرایه ای که 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 آرایه هایست که از نوع `string` تعریف می شوند. دوستانی که با زبان C کار کرده اند حتماً می دانند که آرایه ای از نوع رشته ای در C وجود ندارد و برای نگهداری چندین رشته در یک آرایه باید از آرایه دو بعدی استفاده کرد. در C# این قابلیت فراهم شده تا با استفاده از یک آرایه تک بعدی بتوان چندین رشته را ذخیره نمود بدین صورت که هر یک از عناصر آرایه یک بعدی محلی برای ذخیره سازی یک رشته است و همانند زبان C نیاز به پردازش های گاه پیچیده بر روی آرایه های چند بعدی بمنظور کار با رشته ها، وجود ندارد. بعنوان یک توضیح کمی اختصاصی عرض می کنم که در زبان هایی مانند C، در صورتیکه می خواستید چندین رشته را در آرایه ای ذخیره کنید تا بتوانید با اندیس به آنها دسترسی داشته باشید، مجبور به تعریف یک آرایه دو بعدی بودید که با استفاده از تنها اندیس اول آرایه می توانستید به عناصر رشته ای آرایه دسترسی پیدا کنید، اما در C# تنها با استفاده از یک آرایه تک بعدی می توان همان کار را انجام داد.

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

```
.....
```

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

```
.....
```

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

My name is Meysam

مطلب این درس در اینجا به پایان می رسد. در صورتیکه نیاز دارید تا در مورد عملگرهای زبان C# بیشتر توضیح دهم حتماً ذکر کنید تا در مطلب بعدی توضیح کاملتری در مورد آنها برای شما تهیه کنم. خیلی دوست داشتم که در مورد تمام عملگرهای زبان C# در همین درس توضیح بدهم اما هم فرصت اندک است و هم حجم مطلب این قسمت زیاد می شد و هم اینکه فکر کردم احتمالاً دوستان با این عملگرها آشنایی دارند. نکته دیگری که باید به آن اشاره کنم و اینست که در این سایت سعی شده است تا زبان برنامه نویسی C# به سادگی و به سرعت آموزش داده شود و علت اینکه به برخی از جزئیات تخصصی پرداخته نمی شود نیز همین مطلب می باشد. در آینده در مورد آرایه ها بیشتر صحبت می کنم چون عناصر مفید و سودمندی هستند. امید است پس از

کامل کردن مطالب مقدماتی در این سایت و با همکاری شما عزیزان بتوانم به مطالب پیشرفته‌تری از زبان C# بپردازم. بیان نظرات و پیشنهادات شما چه در زمینه مطالب ارائه شده و چه در زمینه متن ارائه شده به شما از لحاظ سادگی و روانی در درک، مرا در امر بهبود مطالب یاری می‌نماید.

بررسی دستور `if`، دستور `if` ساده، دستور `if-else`، دستور `if-else...else...if`، دستور `switch`، دستور `break`، دستور `case`، دستور `goto`، عملگرهای `AND` و `OR`

درس سوم - دستورالعمل‌های کنترلی و شرطی

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

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

بررسی دستور `if` و انواع مختلف آن

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

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

مثال 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 گروهه
if (myInt < 0)
    Console.WriteLine("Your number {0} is less than zero.", myInt);
//تصمیم‌گیری دوگانه
if (myInt != 0)
{
    Console.WriteLine("Your number {0} is not equal to zero.", myInt);
}
else
{
    Console.WriteLine("Your number {0} is equal to zero.", myInt);
}
//تصمیم‌گیری چندگانه
if (myInt < 0 || myInt == 0)
{
    Console.WriteLine("Your number {0} is less than or equal to zero.", myInt);
}
else if (myInt > 0 && myInt <= 10)
{
    Console.WriteLine("Your number {0} is between 1 and 10.", myInt);
}
else if (myInt > 10 && myInt <= 20)
{
    Console.WriteLine("Your number {0} is between 11 and 20.", myInt);
}
else if (myInt > 20 && myInt <= 30)
{
    Console.WriteLine("Your number {0} is between 21 and 30.", myInt);
}
else
{
    Console.WriteLine("Your number {0} is greater than 30.", myInt);
}
} //Main() پایان متد
} //IfSelect پایان کلاس

```

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 اینجا است، 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 پرانتز را به مقدار عددی تبدیل کرده و آنرا به متغیر دیگری از نوع عددی تخصیص می‌دهد. در این مثال نیز همانطور که دیده می‌شود، `myInput` که از نوع رشته‌ای است در داخل پرانتز قرار گرفته و این مقدار برابر با `myInt` که از نوع `int` است قرار گرفته است. با این کار مقدار عددی رشته ورودی کاربر به متغیر `myInt` تخصیص داده می‌شود. (توضیح کامل تری 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} (boolean expression) if` 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 دستورهای داخل بلوک خود را اجرا می‌نماید (قبلاً توضیح داده شد که دستورهایی 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` بشکل `if(myInt > 0)` است. در صورتیکه مقدار `myInt` بزرگتر از عدد صفر باشد، دستور داخل بلوک `if` اجرا می‌شود و در غیر اینصورت اجرای برنامه به بعد از بلوک `if` منتقل می‌گردد.

دومین دستور `if` در این برنامه بسیار شبیه به دستور اول است، با این تفاوت که در این دستور، دستور اجرایی `if` درون یک بلوک قرار نگرفته است. در صورتیکه بخواهیم با استفاده از دستور `if` تنها یک دستورالعمل اجرا شود، نیازی به استفاده از بلوک برای آن دستورالعمل نمی‌باشد. استفاده از بلوک تنها زمانی ضروری است که بخواهیم از چندین دستور استفاده نماییم.

دستور `if-else`

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

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

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

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

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

در صورتیکه نیاز باشد تا چندین حالت منطقی مورد بررسی قرار گیرد و دستورات مربوط به یکی از آنها اجرا شود، از فرم تصمیم‌گیری چندگانه استفاده می‌نماییم. این نوع استفاده از دستور **if** در اصطلاح به **if** تودرتو (**Nested If**) معروف است چراکه در آن از چندین دستور **if** مرتبط به یکدیگر استفاده شده است. چهارمین دستور **if** در مثال ۱-۳ استفاده از **if** تودرتو را نشان می‌دهد. در این حالت نیز دستور با کلمه کلیدی **if** آغاز می‌گردد. شرطی بررسی شده و در صورت **true** بودن دستورات مربوط به آن اجرا می‌گردد. اما اگر مقدار این عبارت منطقی **false** بود آنگاه شرطهای فرعی دیگری بررسی می‌شوند. این شرطهای فرعی با استفاده از **else if** مورد بررسی قرار می‌گیرند. هر یک از این شرطها دارای عبارات منطقی مربوط به خود هستند که در صورت **true** بودن عبارت منطقی دستورات مربوط به آنها اجرا می‌گردد و در غیر اینصورت شرط بعدی مورد بررسی قرار می‌گیرد. باید توجه کنید که در ساختار **if** تودرتو تنها یکی از حالتها اتفاق می‌افتد و تنها یکی از شرطها مقدار **true** را بازمی‌گرداند. فرم کلی **if** تودرتو بشکل زیر است :

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

عملگرهای **OR** و **AND** (|| و &&)

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

عبارت منطقی $(myInt > 0 \&\& myInt \leq 10)$ حاوی عملگر **AND** شرطی (**&&**) می‌باشد. این عبارت در صورتی مقدار **true** بازمی‌گرداند که هر دو عملوند **AND** دارای مقدار **true** باشند. یعنی در صورتیکه **myInt** هم بزرگتر از صفر باشد و هم کوچکتر از 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;
        } //switch پایان بلوک
        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: short, sbyte, byte, 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 کد و دستورات عملی در بین آنها، پشت سر هم قرار گیرند و دیگری در زمانیکه از دستور goto استفاده شده باشد.

در صورتیکه دو دستور **case** بدون وجود کدی در بین آنها، پشت سر یکدیگر قرار گیرند، بدین معناست که برای هر دو **case** مورد نظر یک عمل خاص در نظر گرفته شده است. به مثال زیر توجه نمایید.

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

در این مثال، همانطور که مشاهده می‌کنید، سه دستور **case** بدون وجود کدی در بین آنها پشت سر یکدیگر قرار گرفته‌اند. این عمل بدین معناست که برای تمامی گزینه‌های 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 بی مورد از دستور **goto** باعث ناخوانا شدن برنامه شده و عیب‌یابی (**Debug**) برنامه را بسیار دشوار می‌نماید. در برنامه‌نویسی‌های امروزی استفاده از دستور **goto** بغیر از موارد بسیار لازم و ضروری منسوخ شده و به هیچ عنوان توصیه نمی‌شود. برای تولید و ساخت حلقه نیز

دستورات مفید و سودمندی در زبان تعبیه شده‌اند که استفاده از **goto** را به حداقل می‌رسانند. دستورات حلقه در مبحث آینده مورد بررسی قرار خواهند گرفت.

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

```
switch(myChar)
{
    case 'A' :
        Console.WriteLine("Add operation\n");
        break;
    case 'M' :
        Console.WriteLine("Multiple operation\n");
        break;
    case 'S' :
        Console.WriteLine("Subtraction operation\n");
        break;
    default :
        Console.WriteLine("Error, Unknown operation\n");
        break;
}
```

معادل بلوک **switch** با استفاده از **if-else**

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

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

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

درس چهارم - دستورات عمل‌های کنترلی، حلقه‌ها

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

- ✓ حلقه *while*
- ✓ حلقه *do-while*
- ✓ حلقه *for*
- ✓ حلقه *foreach*
- ✓ مطالب تکمیلی درباره دستورات عمل *break*
- ✓ فراگیری نحوه بکارگیری دستورات عمل *continue*

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

حلقه *while*

ابتدا به مثال زیر توجه نمایید.

```
using System;
```

```
class WhileLoop
```

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

مثال ۴-۱ که در بالا ملاحظه می‌کنید، یک حلقه *while* ساده را نشان می‌دهد. این حلقه با کلمه کلیدی *while* آغاز شده و سپس به دنبال آن یک عبارت منطقی قرار می‌گیرد و مورد بررسی قرار می‌گیرد. تمامی دستورات عمل‌های کنترلی از یک عبارت منطقی بهره می‌گیرند و این بدین معناست که ابتدا این عبارت باید بررسی شود تا مشخص شود مقدار این عبارت *true* است یا *false*. در این مثال مقدار متغیر *myInt* مورد بررسی قرار می‌گیرد تا چک شود آیا مقدارش از ۱۰ کوچکتر هست یا خیر. چون در ابتدای برنامه به این متغیر مقدار صفر تخصیص داده شده است، عبارت منطقی مقدار *true* را باز می‌گرداند و سپس بلوک قرار گرفته بعد از عبارت منطقی مورد اجرا قرار می‌گیرد.

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

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

حلقه do-while

ابتدا به مثال ۴-۲ توجه نمایید.

```
using System;
```

```
class DoLoop
```

```
{
    public static void Main()
    {
        string myChoice;
        do
        {
            // منویی نمایش داده می‌شود
            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 می‌شود
            myChoice = Console.ReadLine();
            // تصمیمی بر اساس ورودی 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;
}
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 می‌دهد. ساختار نوشتاری این دستور بصورت زیر است :

```

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

```

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

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

یک نگاه کلی به مثال ۴-۲ ببندازیم. در متد *Main()* متغیر *myChoice* را از نوع رشته‌ای تعریف نموده‌ایم. سپس یکسری دستورات را بر روی کنسول چاپ نموده‌ایم. این دستورات منوهای انتخاب برای کاربر هستند. ما باید ورودی از کاربر دریافت کنیم که چون این عمل از طریق *Console.ReadLine()* صورت گرفته، باید در متغیری از نوع رشته‌ای قرار گیرد و از اینرو این ورودی را در *myChoice* قرار داده‌ایم. ما باید ورودی را از کاربر دریافت کنیم و بر روی آن پردازش انجام دهیم. یک روش کارآمد برای این منظور استفاده از دستورالعمل *switch* است. همانطور که در دستور *switch* ملاحظه می‌کنید، بری *default* نیز دستوری در نظر گرفته شده است که نشان می‌دهد 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 می‌دهد. استفاده از حلقه *for* برای زمانی مناسب است که دقیقاً بدانید که حلقه چندبار باید تکرار شود. محتویات درون پرانتزهای حلقه *for* از سه قسمت تشکیل شده است :

(*<initializer list>; <boolean expression>; <postloopaction list>*)

initializer list لیستی از عبارات است که بوسیله کاما از یکدیگر جدا می‌شوند. این عبارات تنها یکبار در طول دوره کاری حلقه *for* پردازش می‌شوند. همانطور که در مثال ۴-۳ 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 ۴-۳ 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 استفاده 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 ۴-۳ 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 هر بار 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، دستورالعملی 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 موقف 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 در س ۴ است، با این تفاوت که در درس چهارم چاپ منو و دریافت ورودی از کاربر در متد `Main()` صورت می‌گرفت در حالیکه در این مثال، این اعمال در یک متد مجزا بنام `getChoice()` صورت می‌گیرد. نوع بازگشتی این متد از نوع رشته‌ای است. از این رشته در دستور `switch` در متد `Main()` استفاده می‌شود. همانطور که ملاحظه می‌نمایید، پرازنزهای متد `getChoice()` خالی هستند، یعنی این متد دارای پارامتر نیست، از اینرو هیچ اطلاعاتی به/از این متد منتقل نمی‌شود.

درون این متد، ابتدا متغیر `myChoice` را اعلان نموده‌ایم. هرچند نام و نوع این متغیر همانند متغیر `myChoice` موجود در متد `Main()` است، اما این دو متغیر دو متغیر کاملاً مجزا از یکدیگر می‌باشند. هر دو این متغیرها، متغیرهای محلی (`Local`) هستند، از اینرو تنها درون بلوکی که تعریف شده‌اند قابل 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
        {
            // منوی نمایش داده شده و ورودی از کاربر دریافت می‌گردد
            myChoice = mp.getChoice();
            // تصمیمی بر اساس ورودی کاربر گرفته می‌شود
            mp.makeDecision(myChoice);
            // جهت دیدن نتایج توسط کاربر، اجرای برنامه موقتاً موقوف می‌گردد
            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 می‌نماید
    } //پایان متد Main
} //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;
} //پایان متد getChoice()
// تصمیم‌گیری
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 ۲-۵، نمونه تغییر یافته مثال ۱-۵ 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 ۲-۵ کلاسی با نام **Address** با دو فیلد از نوع رشته تولید کرده‌ایم.

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

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

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

پارامترهای **out** در مواردی استفاده می‌شوند که ارسال اطلاعات به متد از طریق پارامتر مد نظر نباشد، بلکه ارسال اطلاعات از متد مورد نظر باشد. استفاده از این پارامترها از اینرو کارآمد هستند که برنامه مجبور به کپی کردن پارامتر به متد نیست و از حجم

سرباره (Overhead) برنامه می‌کاهد. در برنامه‌های عادی این مسئله چندان به چشم نمی‌آید، اما در برنامه‌های تحت شبکه که سرعت ارتباط و انتقال داده‌ها بسیار مهم است، این پارامترها ضروری می‌شوند.

متد `modifyAddress()` دارای پارامتری از نوع `out` است. پارامترهای `out` فقط به متد فراخواننده آن بازگشت داده می‌شوند. از آنجائیکه این پارامترها از متد فراخواننده هیچ مقداری دریافت نمی‌کنند و فقط درون متدی که به عنوان پارامتر به آن ارسال شده‌اند قابلیت تغییر دارند، از اینرو درون این متدهایی که به آنها ارسال می‌شوند، قبل از اینکه بتوان از آنها استفاده نمود باید مقداری به آنها تخصیص داده شود. اولین خط در متد `modifyAddress()` بصورت توضیحات نوشته شده است. این خط را از حالت توضیحات خارج کرده و سپس برنامه اجرا کنید تا ببینید چه اتفاقی رخ خواهد داد. هنگامیکه این پارامتر مقدار دهی شود و مقداری را به متد فراخواننده خود بازگرداند، این مقدار بر روی متغیر متد فراخواننده کپی می‌گردد. توجه نمایید که پارامترهای `out` می‌بایست قبل از دستور `return` درون متد مقدار دهی شده باشند.

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

خلاصه

در این درس، با ساختار کلی یک متد آشنا شدید. 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.

```
// Namespace اعلان
using System;

// C# Persian Namespace
namespace csharp-persian
{
    // کلاس آغازین برنامه
    class NamespaceCSS
    {
        // آغاز اجرای برنامه
        public static void Main()
        {
            // چاپ خروجی در کنسول
            Console.WriteLine("This is the new C# Persian Namespace.");
        }
    }
}
```

مثال 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.

```
// Namespace اعلان
using System;

// C# Persian Namespace
namespace csharp-persian
```

```

{
    namespace tutorial
    {
        // کلاس آغازین برنامه
        class NamespaceCSS
        {
            // آغاز اجرای برنامه
            public static void Main()
            {
                // چاپ خروجی در کنسول
                Console.WriteLine("This is the new C# Persian Namespace.");
            }
        }
    }
}

```

Namespace ها این امکان را فراهم می‌آورند 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 می‌آییم، Namespace های تخصصی تر قرار می‌گیرند. این سیستم سلسله مراتبی بوسیله Namespace های تودرتو قابل پیاده‌سازی هستند. این پیاده‌سازی در مثال ۲-۶ نشان داده شده است. با قرار دادند کدهای خود در Namespace های فرعی می‌توانید 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 می‌دهد. همانطور که در این مثال مشاهده می‌نمایید، 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 پیاده‌سازی آسان تر از آن می‌باشد.

فراخوانی اعضای یک Namespace

```
// Namespace اعلان
using System;

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

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

مثال 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 *myExample1* و *myExample2* 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 محدوده‌ایست که متد *myPrint2* فراخوانی شده است، نیازی به آمدن نام *csharp-persian* برای این فراخوانی وجود ندارد، زیرا هر دو کلاس متعلق به یک Namespace، یعنی *csharp-persian* هستند.

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

استفاده از هدایتگر using

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

// کلاس آغازین برنامه
class UsingDirective
{
    // آغاز اجرای برنامه
    public static void Main()
    {
        // فراخوانی عضوی از Namespace
        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 می‌خواهید متدهایی را بدون استفاده از نام کامل آنها، به همراه نام Namespace، استفاده نمایید، می‌توانید 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 نموده‌اید. با استفاده از این هدایتگر می‌توانید 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()* می‌باشد. 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 csTut = cssharp-persian.tutorial.myExample;* امکان استفاده از اعضای این *Namespace* را، بدون نیاز به تایپ نام کامل آنها فراهم می‌نماید. چون در ابتدای برنامه از هدایتگر *using* استفاده نموده‌ایم، در متن برنامه متد *myPrint()* را بصورت *myExample.myPrint()* استفاده نموده‌ایم، در صورتیکه از هدایتگر *using* استفاده نمی‌گردیم برای استفاده از این متد باید آنرا بصورت *cssharp-persian.tutorial.myExample.myPrint()* می‌آوردیم.

استفاده از هدایتگر Alias

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

// کلاس آغازین برنامه
class AliasDirective
{
    // آغاز اجرای برنامه
    public static void Main()
    {
        // فراخوانی عضوی از Namespace
        csTut.myPrint();
        myPrint();
    }
    // متدی که ممکن است تولید ابهام نماید.
    static void myPrint()
    {
        Console.WriteLine("Not a member of cssharp-persian.tutorial.myExample.");
    }
}

// C# Persian Tutorial Namespace
namespace cssharp-persian.tutorial
{
    class myExample
    {
        public static void myPrint()
        {
            Console.WriteLine("This is a member of cssharp-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 می‌توانند در خود داشته باشند که در زیر مشاهده می‌نمایید :

Classes
Structures
Interfaces
Enumerations
Delegates

خلاصه

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

درس هفتم - آشنایی با کلاسها در C#

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

- ✓ پیاده‌سازی سازنده‌ها (Constructors)
- ✓ درک تفاوت بین اعضای نمونه (Instance) و استاتیک (Static)
- ✓ آشنایی با تخریب کننده‌ها (Destructors)
- ✓ آشنایی با اعضای کلاسها

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

یک کلاس با استفاده از کلمه کلیدی `class` که بدنبال آن نام کلاس آمده باشد، اعلان می‌گردد و اعضای این کلاس درون `{ }` اعلان می‌گردند. هر کلاس دارای سازنده‌ای می‌باشد که در هربار ایجاد نمونه‌ای جدید از آن کلاس، بصورت خودکار فراخوانی می‌گردد. هدف از سازنده، تخصیص‌دهی اعضای کلاس در زمان ایجاد نمونه‌ای جدید از کلاس است. سازنده‌ها دارای مقادیر بازگشتی نبوده و همواره نامی مشابه نام کلاس دارند. مثال ۱-۷ نمونه‌ای از یک کلاس را نشان می‌دهد.

// Namespace اعلان

using System;

class OutputClass

{

string myString;

// سازنده

public OutputClass(string inputString)

{

myString = inputString;

}

// متد نمونه

public void printString()

{

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

}

// تخریب کننده

~OutputClass()

{

// روتینی جهت آزادسازی برخی از منابع سیستم

}

}

// کلاس آغازین برنامه

class ExampleClass

{


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

در مثال 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. هر بار که شیء از کلاسی خاص ایجاد می‌کنید، در حقیقت نمونه جدیدی از آن کلاس ایجاد کرده‌اید. متد *Main()* در کلاس *ExampleClass* نمونه جدیدی از *OutputClass* را تحت نام *outCl* ایجاد می‌نماید. می‌توان نمونه‌های متفاوتی از کلاس *OutputClass* را با نامهای مختلف ایجاد نمود. هر یک از این نمونه‌های مجزا بوده و به تنهایی عمل می‌کنند. به عنوان مثال اگر دو نمونه از کلاس *OutputClass* همانند زیر ایجاد نمایید :

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

با این اعلان، شما دو نمونه از کلاس *OutputClass* را ایجاد کرده‌اید که یک از آنها دارای فیلد *myString* و متد *printString()* هستند و این فیلدها و متدها کاملاً از یکدیگر مجزا می‌باشند. به بیان دیگر در صورتیکه عضوی از کلاس استاتیک

باشد از طریق ساختار نوشتاری `<static class member>.<class name>` قابل دسترس خواهد بود. در این مثال نمونه‌ها `oc1` و `oc2` هستند. فرض کنید کلاس `OutPutClass` دارای متد استاتیک زیر باشد :

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

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

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

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

نوع دیگر سازنده‌ها، سازنده‌های استاتیک هستند. از سازنده‌های استاتیک جهت مقداردهی فیلدهای استاتیک یک کلاس استفاده می‌شود. برای اعلان یک سازنده استاتیک تنها کافیست 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. زمانی که 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 واسطها (Interfaces) امکان‌پذیر است که در درسهای آینده به آنها اشاره می‌نماییم.

ChildClass دقیقاً تواناییهای *ParentClass* را دارد. از اینرو می‌توان گفت *ChildClass* یک *ParentClass* است. *ChildClass (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 می‌توانند با کلاس پایه خود ارتباط برقرار نمایند. در مثال ۲-۸ چگونگی انجام این عمل را در سازنده *ChildClass* نشان می‌دهد. استفاده از " : " و کلمه کلیدی *base* باعث فراخوانی سازنده کلاس پایه به همراه لیست پارامترهایش می‌شود. اولین سطر خروجی، فراخوانی سازنده کلاس پایه را به همراه رشته "From Derived" نشان می‌دهد.

ممکن است حالتی رخ دهد که نیاز داشته باشید تا متد موجود در کلاس پایه را خود پیاده‌سازی نمایید. کلاس *Child* این عمل را با اعلان متد *Print()* مربوط به خود انجام می‌دهد. متد *Print()* مربوط به کلاس *Child*، متد *Print()* کلاس *Parent* را پنهان می‌کند. نتیجه این کار آنست که متد *Print()* کلاس *Parent* تا زمانیکه عمل خاصی انجام ندهیم قابل فراخوانی نمی‌باشد.

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

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 پایه‌اش می‌باشد. این مسئله باعث می‌شود تا بتوان کلاس مشتق شده را مورد عمل *Casting* قرار داد و آنرا نمونه‌ای از کلاس پایه‌اش قرار داد. آخرین خط خروجی نشان می‌دهد که متد *Print()* کلاس *Parent* اجرا شده است.

به وجود کلمه کلیدی *new* در متد *Print()* کلاس *Child* توجه نمایید. این عمل باعث می‌شود تا متد *Print()* کلاس *Child* متد *Print()* کلاس پایه‌اش را پنهان نماید. در صورتیکه از کلمه کلیدی *new* استفاده نشود، کامپایلر پیغام خطایی را تولید

می‌کند تا توجه شما را به این مسئله جلب کند. توضیحات بیشتر در این زمینه مربوط به مبحث چندریختی (Polymorphism) است که در درس آینده آنرا بررسی خواهیم نمود.

خلاصه

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

درس نهم _ چند ریختی (Polymorphism)

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

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

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

متد مجازی (Virtual Method)

using System;

public class DrawingObject

{

public virtual void Draw()

{

Console.WriteLine("I'm just a generic drawing object.");

}

}

مثال ۹-۱ کلاس *DrawingObject* را نشان می‌دهد. این کلاس می‌تواند بعنوان کلاسی پایه جهت کلاسهای دیگر در نظر گرفته شود. این کلاس تنها دارای یک متد با نام *Draw()* می‌باشد. این متد دارای پیشوند *virtual* است. وجود کلمه *virtual* بیان می‌دارد که کلاسهای مشتق شده از این کلاس می‌توانند، این متد را *override* نمایند و آنرا به طریقه دلخواه پیاده‌سازی کنند.

```
using System;
```

```
public class Line : DrawingObject
```

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

```
public class Circle : DrawingObject
```

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

```
public class Square : DrawingObject
```

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

در مثال ۹-۲، سه کلاس دیده می‌شود. این کلاسها از کلاس *DrawingObject* ارث‌بری می‌کنند. هر یک از این کلاسها دارای متد *Draw()* هستند و تمامی آنها دارای پیشوند *override* می‌باشند. وجود کلمه کلیدی *override* قبل از نام متد، این امکان را فراهم می‌نماید تا کلاس، متد کلاس پایه خود را *override* کرده و آنرا به طرز دلخواه پیاده‌سازی نماید. متدهای *override* شده باید دارای نوع و پارامترهای مشابه متد کلاس پایه باشند.

پیاده‌سازی چند ریختی

```
using System;
```

```
public class DrawDemo
```

```
{
```

```
    public static int Main( )
```

```
    {
```

```
        DrawingObject[] dObj = new DrawingObject[4];
```

```
        dObj[0] = new Line();
```

```
        dObj[1] = new Circle();
```

```
        dObj[2] = new Square();
```

```
        dObj[3] = new DrawingObject();
```

```
        foreach (DrawingObject drawObj in dObj)
```

```
        {
```

```
            drawObj.Draw();
```

```
        }
```

```
        return 0;
```

```
    }
```

```
}
```

مثال ۹-۳ برنامه‌ای را نشان می‌دهد که از کلاسهای مثال ۹-۱ و ۹-۲ استفاده می‌کند. در این برنامه چند ریختی پیاده‌سازی شده است. در متد `Main()` یک آرایه ایجاد شده است. عناصر این آرایه از نوع `DrawingObject` تعریف شده است. این آرایه `dObj` نامگذاری شده و چهار عضو از نوع `DrawingObject` را در خود نگه می‌دارد.

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

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

I'm a Line.

I'm a Circle.

I'm a Square.

I'm just a generic drawing object.

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

خلاصه

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

درس دهم - ویژگیها در C#

در این درس با ویژگیها (Properties) در زبان C# آشنا خواهیم شد. اهداف این درس به شرح زیر 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 می‌کند. در متد Main() نمونه جدیدی از شی PropertyHolder با نام propHold ایجاد می‌گردد. سپس بوسیله متد setSomeProperty مقدار someMethod از propHold برابر با ۵ می‌گردد و سپس برنامه مقدار property را با استفاده از فراخوانی متد Console.WriteLine() در خروجی نمایش می‌دهد. آرگومان مورد استفاده برای بدست آوردن مقدار property فراخوانی به متد getSomeProperty() است که توسط آن عبارت "Property Value" 5 : در خروجی نمایش داده می‌شود.

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

مثال 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 می‌دهد. کلاس PropertyHolder دارای پیاده‌سازی از ویژگی SomeProperty است. توجه نمایید که اولید حرف از نام ویژگی با حرف بزرگ نوشته شده و این تنها تفاوت میان اسم ویژگی SomeProperty و فیلد someProperty می‌باشد. ویژگی دارای دو accessor با نامهای set و get است. accessor get مقدار فیلد someProperty را باز می‌گرداند. set accessor نیز با استفاده از مقدار value، مقداری را به

someProperty تخصیص می‌دهد. کلمه value که در set accessor آورده شده است جزو کلمات رزرو شده زبان C# می‌باشد.

کلاس PropertyTester از ویژگی someProperty مربوط به کلاس PropertyHolder استفاده می‌کند. اولین خط در متد Main() شی‌ای از نوع PropertyHolder با نام propHold ایجاد می‌نماید. سپس مقدار فیلد someProperty مربوط به شیء propHold، با استفاده از ویژگی SomeProperty به ۵ تغییر می‌یابد و ملاحظه می‌نمایید که مسئله به همین سادگی است و تنها کافی است تا مقدار مورد نظر را به ویژگی تخصیص دهیم.

پس از آن، متد Console.WriteLine() مقدار فیلد someProperty شیء propHold را چاپ می‌نماید. این عمل با استفاده از ویژگی SomeProperty شیء propHold صورت می‌گیرد.

ویژگیها را می‌توان طوری ایجاد نمود که فقط خواندنی (Read-Only) باشند. برای این منظور تنها کافیسست 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 می‌دهد. کلاس `PropertyHolder` دارای ویژگی `SomeProperty` است که فقط `get accessor` را پیاده‌سازی می‌کند. این کلاس `PropertyHolder` دارای سازنده‌ایست که پارامتری از نوع `int` دریافت می‌نماید.

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

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

ویژگیهای فقط نوشتنی (*Write-Only Properties*)

به مثال زیر توجه فرمایید :

مثال 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 سازهٔ 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 - اندیسها در C# (Indexers)

در این درس با اندیکسرها در C# آشنا می‌شویم. اهداف این درس به شرح زیر می‌باشند :

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

اندیکسرها

اندیکسرها مفهومی بسیار ساده در زبان C# هستند. با استفاده از آنها می‌توانید از کلاس خود همانند یک آرایه استفاده کنید. در داخل کلاس مجموعه‌ای از مقادیر را به هر طریقی که مورد نظرتان هست مدیریت کنید. این اشیاء می‌توانند شامل مجموعه‌ای از اعضای کلاس، یک آرایه دیگر، و یا مجموعه‌ای از ساختارهای پیچیده داده‌ای باشند، جدا از پیاده‌سازی داخلی کلاس، داده‌های این ساختارها از طریق استفاده از اندیکسرها قابل دسترسی هستند. به مثالی در این زمینه توجه کنید :

مثال ۱۱-۱ : نمونه‌ای 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 پیاده‌سازی اندیکسر را نشان می‌دهد. کلاس `IntIndexer` دارای آرایه رشته‌ای بنام `myData` می‌باشد. این آرایه، عنصری خصوصی (`private`) است و کاربران خارجی (`external users`) نمی‌توانند به آن دسترسی داشته باشند. این آرایه درون سازنده (`constructor`) کلاس تخصیص‌دهی می‌گردد که در آن پارامتر `size` از نوع `int` دریافت می‌شود، از آرایه `myData` نمونه‌ای جدید ایجاد می‌گردد، سپس هر یک از المانهای آن با کلمه "empty" مقداردهی می‌گردد.

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

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

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

استفاده از `integer` جهت دسترسی به آرایه‌ها در اغلب زبانهای برنامه‌سازی رایج است ولی زبان `C#` چیزی فراتر از آنرا نیز پشتیبانی می‌کند. در `C#` اندیکسرها را می‌توان با چندین پارامتر تعریف کرد و هر پارامتر می‌تواند از نوع خاصی باشد. پارامترهای مختلف بوسیله کاما از یکدیگر جدا می‌شوند. پارامترهای مجاز برای اندیکسر عبارتند از : `integer`، `enum` و `string`. علاوه بر آن، اندیکسرها قابل سرریزی (`Overload`) هستند. در **مثال ۱۱-۲** تغییراتی در مثال قبل ایجاد کرده‌ایم تا برنامه قابلیت دریافت اندیکسرها را نیز داشته باشد.

سرریزی اندیکسرها

مثال ۲-۱۱: اندیکسرهاى سرریز شده (Overloaded Indexers)

```

using System;
/// <summary>
/// پیاده‌سازی اندیکسرهاى 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 بشکل زیر است:

Indexer Output

```

myInd[0]: no value
myInd[1]: no value
myInd[2]: no value
myInd[3]: Another Value
myInd[4]: no value
myInd[5]: Any Value
myInd[6]: no value
myInd[7]: no value
myInd[8]: no value
myInd[9]: Some Value
Number of "no value" entries: 7

```

علت همزیستی هر دو اندیکسر در [مثال 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 پیاده‌سازی آنها آشنا شده‌اید. با استفاده از اندیکسرها می‌توان 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 پیشرفته‌تر 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 می‌کنیم، سرریزی (Overloading) اندیکسرها رخ می‌دهد. در هنگام فراخوانی اندیکسرها، کلاس تنها از روی نوع بازگشتی اندیکسر و تعداد پارامترهای آن متوجه می‌شود که منظور فراخواننده استفاده از کدام اندیکسر بوده است.

۲. از اندیکسر چگونه مانند آرایه استفاده می‌شود؟

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

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

```
public class ListBox: Control
```

```
{
    private string[] items;

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

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

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

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

Csharp-Persian_Indexer_Demo

```
using System;

public class ListBoxTest
{
    // رشته‌های مورد نظر به ListBox تخصیص داده می‌شوند.
    public ListBoxTest(params string[] initialStrings)
```

```

{
    // فضایی را برای 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 اندیکس
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 رشته های موجود را نشان می دهد
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 پیشرفته‌تر و پیچیده‌تر از مطالب قبل به نظر می‌آیند. این انتظار وجود ندارد که شما کلیه مطالب این قسمت را بطور کامل متوجه شده باشید، بلکه هدف تنها آشنا شدن شما با مسایل پیچیده‌تر و واقعی‌تر است. در آینده‌ای نه چندان دور، در سایت به صورت حرفه‌ای کلیه مطالب و سرفصل‌های گفته شده را مورد بررسی قرار خواهیم داد. در ابتدا هدف من

آشنایی شما با کلیه مفاهیم پایه‌ای 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 مثال‌هایی بسیار واقعی و کاربردی خواهیم پرداخت.

درس دوازدهم – ساختارها در C# (Struct)

در این درس با ساختارها (Struct) در زبان C# آشنا می‌شویم. اهداف این درس بشرح زیر می‌باشند

- یک struct یا ساختار (Structure) چیست؟
- پیاده‌سازی ساختارها (Struct)
- استفاده از ساختارها (Struct)
- نکات مهم و مطالب کمکی دربارهٔ struct ها

ساختار (struct) چیست؟

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

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

اعلان و پیاده‌سازی struct

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

مثال ۱-۱۲ : نمونه‌ای از یک ساختار (Struct)

```
using System;
struct Point
{
    public int x;
    public int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point Add(Point pt)
    {
        Point newPt;
```

```

    newPt.x = x + pt.x;
    newPt.y = y + pt.y;
    return newPt;
}
}
/// <summary>
/// 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 جهن 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 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 پیاده‌سازی است. هر کلاس و یا ساختاری که از یک interface ارث‌بری نماید باید تمامی متدهای آنرا پیاده‌سازی کند. درباره interface ها در آینده صحبت خواهیم کرد.

خلاصه :

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

نکات مهم و مطالب کمکی

۱. تفاوت‌های اصلی بین کلاس و ساختار در چیست؟

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

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

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

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

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

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

```
private int hours, minutes, seconds;
}
```

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

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

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

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 کافیست 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 localVariable;
        :
    }
    private Time field;
}
```

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

درس سیزدهم – 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 چشم‌پوشی کرد. (تاثیرات 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 تاثیری 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 تاثیر قرار نمی‌دهد.

از دیدگاه تکنیکی، واسطها بسط مفهومی هستند که از آن به عنوان انتزاع (Abstract) یاد می‌کنیم. در کلاسهای انتزاعی (که با کلمه کلید abstract مشخص 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 پیاده‌سازی بدنه متد. یک interface همچنین می‌تواند دارای فیلدهایی باشد که تمامی آنها static و final هستند. یک interface تنها یک فرم کلی را بدون پیاده‌سازی به نمایش می‌گذارد.

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

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

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

در این مثال نحوه اعلان واسطی با نام IMyInterface نشان داده شده است. یک قاعده (نه قانون!) برای نامگذاری واسطها آنست که نام واسطها را با "I" آغاز کنیم که اختصار کلمه interface است. در interface این مثال تنها یک متد وجود دارد. این متد می‌توان هر متدی با انواع مختلف پارامترها و نوع بازگشتی باشد. توجه نمایید همانطور که گفته شد این متد دارای پیاده‌سازی نیست و تنها اعلان شده است. نکته دیگر که باید به آن توجه کنید آنست که این متد به جای داشتن {} به عنوان بلوک خود، دارای ; در انتهای اعلان خود می‌باشد. علت این امر آنست که interface تنها نوع بازگشتی و پارامترهای متد را مشخص می‌نماید و کلاس یا شی‌ای که از آن ارث می‌برد باید آنرا پیاده‌سازی نماید. مثال زیر نحوه استفاده از این واسط را نشان می‌دهد.

مثال ۱-۱۳: استفاده از واسطها و ارث‌بری از آنها

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

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

مثال ۲-۱۳: ارث‌بری واسطها از یکدیگر

```
using System;

interface IParentInterface
{
    void ParentInterfaceMethod();
}

interface IMyInterface : IParentInterface
{
    void MethodToImplement();
}

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

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

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

مثال ۲-۱۳ دارای ۲ واسط است: یکی `IMyInterface` و واسطی که از آن ارث می‌برد یعنی `IParentInterface`. هنگامیکه واسطی از واسط دیگری ارث‌بری می‌کند، کلاس یا ساختاری که این واسطها را پیاده‌سازی می‌کند، باید تمامی اعضای واسطهای موجود در سلسله مراتب ارث‌بری را پیاده‌سازی نماید. در مثال ۲-۱۳، چون کلاس `InterfaceImplementer` از واسط `IMyInterface` ارث‌بری نموده، پس از واسط `IParentInterface` نیز ارث‌بری دارد، از اینرو باید کلیه اعضای این دو واسط را پیاده‌سازی نماید.

چند نکته مهم:

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

attributes: صفتهای واسط

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

interface-name: نام واسط

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

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

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

- ۴- هدف از ایجاد یک interface تعیین توانایی‌هاست که می‌خواهیم در یک کلاس وجود داشته باشند.

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

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

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

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

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

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

حال بعنوان طراح برنامه، شما وظیفه داری تا به پیاده‌سازی این واسط پردازید، بطوریکه کلیه نیازهای شما را برآورده نماید. نمونه‌ای از این پیاده‌سازی در مثال ۳-۱۳ آورده شده است.

مثال ۳-۱۳: پیاده‌سازی واسط و ارث‌بری - مثال کاربردی

```
using System;

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

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

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

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

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

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

```

IStorable isDoc = (IStorable) doc;
isDoc.Status = 0;
isDoc.Read( );
Console.WriteLine("IStorable Status: {0}", isDoc.Status);
}
}

```

خروجی برنامه نیز بشکل زیر است :

Output:

```

Creating document with: Test Document
Implementing the Read Method for IStorable
Document Status: -1
Implementing the Read Method for IStorable
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 دقیقا همان گونه که در واسط مشخص شده پیاده سازی نماید. به بیان کلی، کلاسی که از یک واسط ارث می برد، فرم و ساختار کلی خود را از واسط می گیرد و نحوه رفتار و پیاده سازی آنرا خود انجام می دهد.

خلاصه :

در این درس با مفاهیم کلی و اصلی درباره واسطها آشنا شدید. هم اکنون می دانید که واسطها چه هستند و سودمندی استفاده از آنها چیست. همچنین نحوه پیاده سازی واسط و ارث بری از آنرا آموختید.

مبحث واسطها بسیار گسترده و مهم است و امید است در بخشهای آینده در سایت، بتوانم تمامی مطالب را بطور حرفه ای و کامل در اختیار شما قرار دهم.

درس چهاردهم - 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.

مقدمه‌ای بر رخدادها و delegate ها

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

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

رخدادها (Events) بیان این مفهوم هستند که در صورت اتفاق افتادن عملی در برنامه، کاری باید صورت گیرد. در زبان C# مفاهیم Event و Delegate دو مفهوم بسیار وابسته به یکدیگر هستند و با یکدیگر در تعامل می‌باشند. برای مثال، مواجهه با رخدادها و انجام عمل مورد نظر در هنگام اتفاق افتادن یک رخداد، نیاز به یک event handler دارد تا در زمان بروز رخداد، بتوان به آن مراجعه نمود. Event handler ها در C# معمولاً با delegate ها ساخته می‌شوند.

از delegate، می‌توان به عنوان یک Callback یاد نمود، بدین معنا که یک کلاس می‌تواند به کلاسی دیگر بگوید: "این عمل خاص را انجام بده و هنگامیکه عملیات را انجام دادی منرا نیز مطلع کن". با استفاده از delegate ها، همچنین می‌توان متدهایی تعریف نمود که تنها در زمان اجرا قابل دسترسی باشند.

Delegate

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

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

شیء یا متد خاص، آنرا به یک delegate مرتبط نمود و سپس آن delegate را به متد یا شیء خاصی در هنگام اجرای برنامه متصل نمود.

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

روش دیگر حل مسئله آنست که، می‌توان delegate ی را به الگوریتم مورد نظر ارسال نمود و اجازه داد تا متد موجود در آن، عمل مورد نظر ما را انجام دهد. چنین عملی در مثال 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();
        // ساخت نمونه‌ای جدید از delegate
        Comparer cmp = new Comparer(Name.CompareFirstNames);
        Console.WriteLine("\nBefore Sort: \n");
        sd.PrintNames();

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

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

```

```

    {
        temp = names[i];
        names[i] = names[j];
        names[j] = (Name)temp;
    }
}
}
}
public void PrintNames()
{
    Console.WriteLine("Names: \n");
    foreach (Name name in names)
    {
        Console.WriteLine(name.ToString());
    }
}
}

```

اولین اعلان در این برنامه، اعلان `delegate` است. اعلان `delegate` بسيا رشبیه به اعلان متد است، با این تفاوت که دارای کلمه کلیدی `delegate` در اعلان است و در انتهای اعلان آن ";" قرار می‌گیرد و نیز پیاده‌سازی ندارد. در زیر اعلان `delegate` که در مثال ۱-۱۴ آورده شده را مشاهده می‌نمایید:

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

این اعلان، مدل متدی را که `delegate` می‌تواند به آن اشاره کند را تعریف می‌نماید. متدی که می‌توان از آن بعنوان `delegate handler` برای `Comparer` استفاده نمود، هر متدی می‌تواند باشد اما حتماً باید پارامتر اول و دوم آن از نوع `object` بوده و مقداری از نوع `int` بازگرداند. در زیر متدی که بعنوان `delegate handler` در مثال ۱-۱۴ مورد استفاده قرار گرفته است، نشان داده شده است:

```

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

```

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

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

در مثال 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 می‌توان `handler` دیگری با نام `CompareLastNames()` تعریف کنید، نمونه جدیدی از `Comparer` را با این پارامتر ایجاد کرده و سپس آنرا به متد `Sort()` ارسال نمایید.

درک سودمندی *delegate* ها

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

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 پیاده‌سازی نمایید و سپس با استفاده از کلاسی دیگر، این متد را هر یک ثانیه یکبار فراخوانی نمائیم. به مثال زیر توجه کنید :

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

```

{
    Ticker pulsed = new Ticker();
    Subscriber worker = new Subscriber();
    pulsed.Attach(worker);
    :
}
}

```

این مثال مطمئناً کار خواهد کرد اما ایدآل و بهینه نیست. اولین مشکل آنست که کلاس Ticker بشدت وابسته به Subscriber است. به بیان دیگر تنها نمونه‌های جدید کلاس Subscriber می‌توانند از کلاس Ticker استفاده نمایند. اگر در برنامه کلاس دیگری داشته باشید که بخواهید آن کلاس نیز هر یک ثانیه یکبار اجرا شود، می‌بایست کلاس جدیدی شبیه به Ticker ایجاد کنید. برای بهینه کردن این مسئله می‌توانید از یک واسط (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 نمی‌کرد :

```

interface Tickable
{
    void Tick();
}

```

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

```

delegate void Tick();

```

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

```

void Example(Tick param)
{
    :
}

```

فراخوانی delegate

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

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

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

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

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

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

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

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

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

با استفاده از delegate می‌توانید کلاس Ticker را پیاده‌سازی کنید :

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

```
{
    :
    public void Attach(Tick newSubscriber)
    {
        subscribers.Add(newSubscriber);
    }
    public void Detach(Tick exSubscriber)
    {
        subscribers.Remove(exSubscriber);
    }

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

```

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

```

ساخت نمونه‌های جدید از یک delegate

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

```

class Clock
{
    :
    public void RefreshTime(int hours, int minutes, int 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 می‌دهیم تا متدی را که بهنگام روی دادن رخداد فراخوانی می‌شود، معین گردد. مثال زیر روش تعریف رخداد را نشان می‌دهد.

مثال ۲-۱۴: اعلان و پیاده‌سازی رخدادها

```

using System;

public delegate void MyDelegate();

```

```

class Listing14-2
{
    public static event MyDelegate MyEvent;

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

        // فراخوانی رخداد
        MyEvent ();

        Console.ReadLine ();
    }

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

```

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

مثال فوق را بدون استفاده از رخداد نیز می‌توان نوشت. این نسخه از مثال ۱۴-۲ 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 (تبث شدن در یک رخداد)

برای افزودن delegate جدید به یک رخداد کافیهست تا از عملگر += استفاده نماییم. مثال زیر کلاس Clock را نشان می‌دهد که در آن فیلدی از نوع Ticker با نام pulsed وجود دارد. کلاس Ticker دارای رخداد tick از نوع delegate ی بنام Tick است. متد Clock.Start 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 پرانتز فراخوانی می‌گردد. پس از اینکه رخدادی فراخوانی شد، کلیه `delegate` های مرتبط با آن بترتیب فراخوانی می‌شوند. برای مثال در اینجا کلاس `Ticker` را در نظر بگیرید که دارای متد `private` با نام `Notify` است که رخداد `tick` را فرا می‌خواند:

```

class Ticker
{

```

```

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

```

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

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

```

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

```

مثالی پیشرفته از استفاده رخدادها در فرمهای ویندوز

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

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

```
namespace System
```

```

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

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

```

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

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

```

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

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

```

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

این کد، شبیه به کدهایی است که توسط محیط‌های برنامه‌سازی نظیر Visual Studio.Net و یا C#Builder بطور خودکار تولید می‌شوند و تنها کافیسست تا شما کد مربوط به Okay_Click را درون آن وارد نمایید.

رخدادهایی که توسط کلاسهای GUI تولید می‌شوند همواره از یک الگوی خاص پیروی می‌کنند. این رخدادها همواره از نوع delegate ی 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 استثنایا (Exception Handling)

در این درس با چگونگی برخورد با استثنایا (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 استثنایا بوسیله بلوک try/catch
- (۳) آزادسازی منابع تخصیص داده شده به یک برنامه در یک بلوک finally

استثنایا، 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 خطاهای 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 استثنایا (Exception Handling) مطرح می‌شود.

هنگامیکه استثنایی رخ می‌دهد، در اصطلاح می‌گوئیم که این استثناء، thrown شده است. در حقیقت thrown، شیء‌ای 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 استثنایا 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 استثنایایی، دقیقاً با روتین‌های نوشته شده توسط برنامه‌نویس در ارتباط هستند و بستگی کامل به الگوریتمی دارد که وی برای چنین شرایطی در نظر گرفته است. برای مثال، در صورتیکه با استفاده از متد System.IO.File.OpenRead()، اقدام به باز کردن فایلی نماییم، احتمال وقوع (Thrown) یکی از استثنایای زیر وجود دارد :

```
SecurityException
ArgumentException
ArgumentNullException
PathTooLongException
DirectoryNotFoundException
UnauthorizedAccessException
FileNotFoundException
```

NotSupportedException

با نگاهی بر مستندات Net Framework SDK، به سادگی می‌توان از خطاها و استثناهایی که ممکن است یک متد ایجاد کند، مطلع شد. تنها کافیست به قسمت Reference/Class Library رفته و مستندات مربوط به Namespace/Class/Method را مطالعه نمایید. در این مستندات هر خطا دارای لینکی به کلاس تعریف کننده خود است که با استفاده از آن می‌توان متوجه شد که این استثناء به چه موضوعی مربوط است. پس از اینکه از امکان وقوع خطایی در قسمتی از برنامه مطلع شدید، لازم است تا با استفاده از مکانیزمی صحیح به مقابله با آن بپردازید.

هنگامیکه یک استثناء در اصطلاح throw 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 استثناهای 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 می‌شود](#).

نکته : توجه نمایید که کلیه مثالهای موجود در این درس به طور تعمیدی دارای خطاهایی هستند تا شما با نحوه مقابله با استنهاها آشنا شوید.

```
using System;
using System.IO;

class TryCatchDemo
{
    static void Main(string[] args)
    {
        try
        {
            File.OpenRead("NonExistentFile");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```

هر چند کد موجود در مثال ۱-۱۵ [تنها دارای یک بلوک catch است](#)، اما تمامی استثناهایی که ممکن است رخ دهند را نشان داده و مورد بررسی قرار می‌دهد زیرا از نوع کلاس پایه استثناء، یعنی Exception تعریف شده است. در کنترل و مقابله با استنهاها، باید استثناهای خاص را زودتر از استنهاهای کلی مورد بررسی قرار داد. کد زیر نحوه استفاده از چند بلوک catch را نشان می‌دهد :

```
catch (FileNotFoundException fnfex)
```

```

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

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

```

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

استثناهایی که مورد بررسی قرار نگیرند، در بالای `Stack` نگهداری می‌شوند تا زمانی که بلوک `try/catch` مناسبی مربوط به آنها یافت شود. در صورتیکه برای استثناء رخ داده بلوک `try/catch` در نظر گرفته نشده باشد، برنامه متوقف شده و پیغام خطایی ظاهر می‌گردد. این چنین حالتی بسیار نامناسب بوده و کاربران را دچار آشفتگی خواهد کرد. استفاده از روشهای مقابله با استثناء در برنامه، روشی مناسب و رایج است و باعث قدرتمند تر شدن برنامه می‌شود.

یکی از حالت‌های بسیار خطرناک و نامناسب در زمان وقوع استثناء، 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 استثناء مناسب است ولی در مورد گفته شده نمی‌تواند کمکی به حل مشکل نماید. برای چنین شرایطی که نیاز به آزادسازی منابع تخصیص داده شده به یک برنامه داریم، از بلوک `finally` استفاده می‌کنیم.

کد نشان داده شده در مثال ۲-۱۵، به خوبی روش استفاده از بلوک `finally` را نشان می‌دهد. همانطور که حتماً می‌دانید، رشته های فایلی پس از اینکه کار با آنها به اتمام می‌رسد باید بسته شوند، در غیر اینصورت هیچ برنامه دیگری قادر به استفاده از آنها نخواهد بود. در این حالت، رشته فایلی، منبعی است که می‌خواهیم پس از باز شدن و اتمام کار، بسته شده و به سیستم باز گردد. در مثال ۲-۱۵، `outStream` با موفقیت باز می‌شود، بدین معنا که برنامه `handle` ای به یک فایل باز شده در اختیار دارد. اما زمانی که می‌خواهیم `inStream` را باز کنیم، استثناء `FileNotFoundException` رخ داده و باعث می‌شود که کنترل برنامه سریعاً به بلوک `catch` منتقل گردد.

در بلوک `catch` می‌توانیم فایل `outStream` را ببندیم. اما برنامه تنها زمانی به بلوک `catch` وارد می‌شود 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 نمی‌شود. بنابراین باید برای بستن این فایل نیز فکری کرد. این کاری است که در بلوک `finally` رخ می‌دهد. 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 می‌توان گفت بلوک `finally`، بلوکی است که تضمین می‌نماید در هر شرایطی اجرا خواهد شد. پس برای حصول اطمینان از اینکه منابع مورد استفاده برنامه پس از خروج برنامه، به سیستم باز گردانده می‌شوند، می‌توان از این بلوک استفاده کرد.

```
using System;
```

```

using System.IO;

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

```

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

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

درس شانزدهم - استفاده از صفتها در C#

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

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

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

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

مفاهیم اولیه درباره صفتها

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

[ObsoleteAttribute]

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

[Obsolete]

همچنین صفتها می‌توانند دارای پارامتر نیز باشند که با استفاده از آنها خواص بیشتری را در اختیار برنامه قرار می‌دهند. در [مثال ۱-۱۶](#) موارد متنوعی از استفاده صفت *ObsoleteAttribute* را مشاهده می‌نمایید.

مثال ۱-۱۶ : 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 می‌افزاید که آن را با دو اعلان قبلی متمایز می‌نماید. نتیجه هر سه اعلان این صفت در زیر آورده شده است. این پیامها، پیامهای کامپایلر **C#** هستند که به هنگام کامپایل برنامه تولید شده‌اند.

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

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

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

[مثال ۱۶-۱](#) شامل صفت دیگری نیز می‌باشد. این صفت **STAThreadAttribute** است که معمولاً در ابتدای کلیه برنامه‌های **C#** و قبل از آغاز متد **Main()** قرار می‌گیرد. این صفت بیان می‌دارد که برنامه **C#** مورد نظر می‌تواند با کد مدیریت نشده **COM** از طریق **Simple Threading Apartment** ارتباط برقرار نماید. استفاده از این صفت در هر برنامه‌ای می‌تواند مفید باشد، چراکه شما بعنوان برنامه نویس هیچ‌گاه اطلاع ندارید که آیا کتابخانه ثالثی که از آن استفاده می‌کنید، قصد برقراری ارتباط با **COM** را دارد یا نه؟ (در صورتیکه با برخی از اصطلاحات بکار رفته آشنایی ندارید اصلاً نگران نشوید. در اینجا هدف تنها نشان دادن موارد استفاده از صفتهاست.)

صفتها می‌توانند دارای چندین پارامتر باشند. در [مثال ۱۶-۲](#)، استفاده از دو پارامتر برای یک صفت نشان داده شده است.

مثال ۱۶-۲

```
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 ۱۶-۲](#) مشاهده می‌کنید، صفت مورد استفاده دارای دو پارامتر است. پارامتر اول که یک جمله متنی است و همانند [مثال ۱۶-۱](#) عمل می‌کند. پارامتر دوم نیز بیان‌کننده نوع پیغامی است که این صفت در هنگام کامپایل تولید می‌کند. در صورتیکه این مقدار برابر با **True** باشد، بدین معناست که در هنگام کامپایل پیغام خطا تولید می‌شود و کامپایل برنامه متوقف می‌گردد. در حالت پیش فرض مقدار این پارامتر برابر با **False** است که بیان می‌دارد، 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 می‌شود.

```
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 می‌گیرد، توسط سایر برنامه‌ها نیز قابل تفسیر و استفاده است.

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

همانطور که در بالا نیز اشاره شد، صفتها می‌توانند دارای پارامتر نیز باشند. این پارامترها به دو دسته تقسیم می‌شوند. پارامترهای محلی (*positional*) و پارامترهای اسمی (*named*). از پارامترهای *positional* در زمانی استفاده می‌شود که می‌خواهیم 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 می‌باشد که ما آنرا در [مثال ۱۶-۱](#) لحاظ نکردیم. همانطور که در [مثال ۱۶-۲](#) مشاهده کردید، از این پارامتر *positional* می‌توان برای ایجاد یک خطا در زمان کامپایل برنامه استفاده نمود.

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

تفاوت پارامترهای *positional* با پارامترهای *named* در آنست که، پارامترهای *named* با نامشان مورد استفاده قرار می‌گیرند و همیشه اختیاری هستند. در [مثال ۱۶-۳](#) صفت **DllImport** را مشاهده می‌نمایید که دارای هر دو نوع پارامتر *named* و *positional* است.

مثال ۱۶-۳

```
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 ۱۶-۳](#) دارای یک پارامتر `positional` ("`User32.dll`") و یک پارامتر `named` ("`EntryPoint=MessageBox`") است. پارامترهای `named` در هر مکانی می‌توانند قرار گیرند و مانند پارامترهای `positional` دارای محدودیت مکانی نیستند. بدین معنا که چون در پارامترهای `named`، نام پارامتر مستقیماً مورد استفاده قرار می‌گیرد، محل قرار گیری آن در لیست پارامترهای صفت مهم نیست اما در مورد پارامترهای `positional` چون اسم پارامتر مورد استفاده قرار نمی‌گیرد، این پارامترها حتماً باید در مکانهای تعیین شده و تعریف شده در لیست پارامترهای صفت قرار گیرند. توجه کنید که چون هدف ما تنها آشنایی با صفتها و نحوه استفاده از آنهاست، درباره پارامترهای مختلف صفت `DllImport` بحث نخواهیم کرد چراکه پارامترهای این صفت نیاز به آشنایی کامل با `Win32 API` دارد.

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

Target های صفتها (عناصری که صفتها بر روی آنها اعمال می‌شوند)

صفتهایی که تا کنون مشاهده کردید، همگی بر روی متدها اعمال شده بودند. اما عناصر مختلف دیگری در `C#` وجود دارند که می‌توان صفتها را بر روی آنها اعمال نمود. جدول ۱-۱۶ عناصر مختلف زبان `C#` را که صفتها بر روی آنها اعمال می‌شوند را نشان می‌دهد.

عناصر اعمال شونده	قابل اعمال به
all	به تمامی عناصر قابل اعمال هستند.
assembly	به تمام یک اسمبلی
class	کلاسها
constructor	سازنده‌ها
delegates	Delegate ها
enum	عناصر شمارشی
event	رخدادها
field	فیلدها
interface	واسطها

method	متدها
module	ماژولها (کدهای کامپایل شده‌ای که می‌توانند به عنوان قسمتی از یک اسمبلی در نظر گرفته شوند).
parameter	پارامترها
property	Property ها
returnvalue	مقادیر بازگشتی
struc	ساختارها

هر چند ممکن است استفاده از این **Target** ها باعث ایجاد ابهام شوند، اما می‌توان با استفاده از این **Target** ها معین کرد که صفت دقیقا به عنصر مورد نظر اعمال شود. یکی از صفت‌هایی که بر روی اسمبلی اعمال می‌شود و باعث ارتباط با **CLS** می‌گردد، صفت **CLSCompliantAttribute** است. **CLS** یا همان **Common Language Specification** امکان برقراری ارتباط بین کلیه زبان‌هایی که تحت **.Net** کار می‌کنند را فراهم می‌نماید. **Target** های صفتها با استفاده از اسم **Target** که بعد از آن کولون قرار می‌گیرد، ایجاد می‌شوند. در مثال 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 کاراتر 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** می‌توان قوانین ارث‌بری این صفت را کنترل نمود. با استفاده از این **property** می‌توان مشخص کرد که آیا کلاسی که از کلاسی که صفت بر روی آن اعمال شده، ارث بری می‌کند نیز، صفت بر روی آن اعمال می‌شود یا نه و یا به عبارتی صفت در کلاس مشتق شده نیز مورد ارث‌بری قرار می‌گیرد یا نه.

حال با استفاده از موارد گفته شده در بالا، می‌خواهیم 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 می‌کند که صفت **Help** تنها بر روی کلاسها قابل اعمال است و در صورتیکه از آن بر روی عنصری به غیر از کلاس استفاده نماییم خطایی رخ خواهد داد. بنابراین کد زیر، خطایی تولید خواهد کرد :

```
[Help("this is a do-nothing class")]
public class AnyClass
{
    [Help("this is a do-nothing method")] //error
    public void AnyMethod()
    {
    }
}
```

و کد خطای تولید شده بشکل زیر خواهد بود :

AnyClass.cs: Attribute 'Help' is not valid on this declaration type.
It is valid on 'class' declarations only.

توجه کنید که با استفاده از **AttributeTargets.All** به صفت **Help** این امکان را می‌دهیم تا بر روی تمامی عناصر موجود اعمال شود. لیست کامل عناصر مجاز نیز بشرح زیر است :

- Assembly,
- Module,
- Class,
- Struct,
- Enum,

- Constructor,
- Method,
- Property,
- Field,
- Event,
- Interface,
- Parameter,
- Delegate,
- All = Assembly , Module , Class , Struct , Enum , Constructor , Method , Property , Field , Event , Interface , Parameter , Delegate,
- ClassMembers = Class , Struct , Enum , Constructor , Method , Property , Field , Event , Delegate , Interface

حال به `AllowMultiple = false` توجه نمایید. با استفاده از این کد، به صفت `Help` اجازه می‌دهیم تا تنها یکبار بر روی عنصری از برنامه اعمال شود. پس کد زیر تولید خطا می‌نماید :

```
[Help("this is a do-nothing class")]
[Help("it contains a do-nothing method")]
public class AnyClass
{
    [Help("this is a do-nothing method")]           //error
    public void AnyMethod()
    {
    }
}
```

و کد خطای تولید شده نیز بصورت زیر است :

AnyClass.cs: Duplicate 'Help' attribute

در نهایت نیز به بررسی `Inherited` می‌پردازیم. با استفاده از این ویژگی، معین می‌کنیم در صورتیکه کلاس دیگری بخواهد از روی کلاسی که صفت بر روی آن اعمال شده ارث‌بری نماید، آیا این صفت بر روی آن کلاس نیز اعمال شود یا نه. در صورتیکه مقدار این ویژگی برابر با `True` باشد، کلاس مشتق شده نیز از صفت ارث‌بری می‌نماید. برای یک مثال می‌توانیم حالت زیر را در نظر بگیریم :

```
[Help("BaseClass")]
public class Base
{
}

public class Derive : Base
{
}
```

تمامی حالت‌های مختلف ترکیب این سه ویژگی بصورت زیر است :

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false,
Inherited = false )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true,
Inherited = false )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false,
Inherited = true )
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true,
Inherited = true )
```

استفاده از پارامترهای **Named** و **Positional** در صفتهای شخصی

همانطور که در قبل نیز اشاره شد، پارامترهای **Positional** پارامترهای سازنده صفت هستند و در هر بار استفاده از صفت باید لحاظ شوند. حال برای بررسی می‌خواهیم پارامترهایی به صفت **Help** خود اضافه نماییم.

مثال 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 `Metadata` مربوط به یک اسمبلی، لیست کاملی از کلاسها، انواع و متدهایی را که برای آن اسمبلی خاص تعریف شده‌اند را، بدست آوریم. به مثال 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 می‌توان ساعتها در مورد صفتهای مختلف بحث نمود. اما آنچه مسلم است تمرین مستمر و پیگیری برنامه نویسی در یافتن مواردی که می‌توان با استفاده از صفتها، برنامه‌ای پویاتر ایجاد نمود، مهمترین عامل در فهم و درک کامل مبحث خواهد بود. درباره Reflection نیز در آینده در یان سایت مفصلا توضیح خواهیم داد.

خلاصه :

در این درس با صفتها آشنا شدید. یاد گرفتید که چگونه از صفتهای موجود در .Net Framework، استفاده کرده و همچنین چگونه صفتهای شخصی و دلخواه خود را تولید نمایید. همچنین با پارامترهای صفتها و انواع آنها و نیز، عناصری که صفتها بر روی آنها اعمال می‌شوند آشنا شدید. در انتهای درس نیز به مختصر درباره Reflection و چگونگی استفاده از صفتها در زمان اجرا صحبت کردیم. امیدوارم مفید واقع شده باشد.

درس هفدهم: انواع شمارشی در C#

در این درس با انواع شمارشی (Enumerator Types) در زبان C# آشنا خواهیم شد. مطالب مورد بررسی در این درس به شرح زیر می باشد :

درک و فهم یک نوع شمارشی یا یک enum
ساخت یک نوع شمارشی جدید
چگونگی استفاده از انواع شمارشی
آشنایی با متدهای مختلف موجود در System.Enum

enum فرم خاصی از انواع مقداری (Value Type) است که از System.Enum مشتق شده و امکان پیمایش درون مجموعه‌ای مشخص را با استفاد از اعداد صحیح برای ما فراهم می نماید. با استفاده از enum می توان مجموعه‌ای از مقادیر ثابت را تعریف نمود که این مقادیر ثابت با استفاده از یک عدد صحیح قابل دسترسی هستند.

استفاده از enum در برنامه‌ها باعث بالا رفتن خوانایی برنامه می شود، چراکه با استفاده از آنها می توان با مجموعه‌ای از اعداد صحیح ترتیبی (Sequential)، با عناوینی تعریف شده، کار کرد. برای مثال، در اعلان یک enum، ما مجموعه‌ای از نامهای مورد نظر را تعریف می نماییم و در برنامه می توانیم از این نامها بصورت ترتیبی استفاده نماییم. Enum به هر یک از عناصر موجود در این مجموعه عددی را تخصیص می دهد که شروع این عدد می تواند توسط برنامه نویس نیز معین گردد. سپس با استفاده از نام عناصر موجود در enum و یا با استفاده از اعدادی که به هر یک از این عناصر تخصیص داده شده، می توان enum را پیمایش نمود و به عناصر آن دسترسی داشت.

همانطور که گفته شد، enum یک نوع مقداری (Value Type) است، از اینرو ارث بری در مورد آن معنایی ندارد. مساوی قرار دادن دو enum نیز، مقادیر یکی را در دیگری کپی می کند. همانطور که در این درس، و در منابع دیگر، خواهید یافت، دسترسی به انواع شمارشی در C# با استفاده از دو کلمه enum و Enum امکان پذیر است. در C# نوع شمارشی enum از نوع BCL خود یعنی Enum ارث بری می کند ! با استفاده از enum یک نوع شمارشی جدید تولید می شود و با استفاده از Enum، می توان به پیاده سازی متدهای استاتیک انواع شمارشی پرداخت.

ایجاد یک نوع شمارشی

BCL Net Framework حاوی enum ها و مثالهای متعددی از استفاده آنها می باشد. برای مثال هرگاه که از MessageBox بر روی فرمی استفاده می شود، می توان از MessageBoxIcon که یک نوع شمارشی است استفاده نمود.

علاوه بر انواع شمارشی تعریف شده و موجود در Net Framework،، زمانهایی نیز وجود دارند که می خواهیم مجموعه‌ای از عناصر را تعریف کرده و از آنها استفاده نماییم. برای دسترسی به عناصری از نوع صحیح، استفاده از enum باعث خوانا شدن برنامه می گردد.

نحوه اعلان یک enum در حالت کلی بصورت زیر است :

```
<modifier> enum <enum_name>
{
```

```

        // Enumeration list
    }

```

در مثال ۱-۱۷ که در زیر ملاحظه می‌کنید، نحوه اعلان و استفاده از `enum` مشخص شده است.

مثال ۱-۱۷: نحوه اعلان یک *enum*

```

using System;

// declares the enum
public enum Volume
{
    Low,
    Medium,
    High
}

// demonstrates how to use the enum

class EnumSwitch
{
    static void Main()
    {
        // create and initialize
        // instance of enum type
        Volume myVolume = Volume.Medium;

        // make decision based
        // on enum value
        switch (myVolume)
        {
            case Volume.Low:
                Console.WriteLine("The volume has been turned Down.");
                break;
            case Volume.Medium:
                Console.WriteLine("The volume is in the middle.");
                break;
            case Volume.High:
                Console.WriteLine("The volume has been turned up.");
                break;
        }
        Console.ReadLine();
    }
}

```


در مثال ۱-۱۷ نمونه‌ای از اعلان یک `enum` را مشاهده می‌نمایید. همانطور که دیده می‌شود، اعلان یک نوع شمارشی با استفاده از کلمه کلیدی `enum` صورت گرفته و سپس به دنبال آن نام این مجموعه مشخص می‌شود. درون کروشه‌های باز و بسته `{ }` نیز، عناصر نوع شمارشی اعلان می‌گردند.

نوع شمارشی تولید شده در این مثال از نوع `Volume` است و در متد `Main()` از آن برای اعلان `myVolume` استفاده شده است. از آنجائیکه `enum` یک نوع مقداری است، می‌توانیم بطور مستقیم آنرا مقداردهی نماییم. پس از آنکه متغیر `myVolume` مقداردهی شد، می‌توان همانند سایر انواع مقداری، مانند `int`، از آن استفاده نمود. در حلقه `switch`، متغیر `myVolume` با عناصر `enum` مقایسه می‌شوند.

در هربار استفاده از عناصر `enum` تولید شده، از نام `enum` تولید شده، در اینجا `Volume`، در ابتدای نام عنصر استفاده می‌نماییم (`Volume.Medium`)، زیرا در صورتیکه در یک برنامه چندین `enum` وجود داشته باشند که دارای عناصری با نامهای یکسان باشند، در صورت عدم استفاده از نام `enum` مورد نظر قبل از عنصر، ابهام بوجود آمده و دچار مشکل می‌شویم.

بطور پیش فرض، به اولین عنصر تعریف شده در `enum` مقدار صفر تخصیص داده می‌شود که این مقدار تنها بعنوان اندیسی جهت دسترسی به این عنصر در نظر گرفته می‌شود. سایر عناصر `enum` نیز بطور صعودی مقدار می‌گیرند و به هر عنصر یک واحد افزوده می‌شود. در مثال ۱-۱۷، عنصر `Low` دارای اندیس صفر بوده و سایر عناصر به ترتیب مقدار ۱ و ۲ خواهند داشت.

در `C#`، برای موارد خاص می‌توان مقادیر پیش فرض در نظر گرفته شده برای عناصر `enum` را تغییر داد. به مثال زیر توجه کنید.

```
enum Months
{
    jan, feb, mar, apr
}

enum Months
{
    jan = 10, feb = 20, mar = 30, apr=40
}
```

همانطور که مشاهده می‌شود، در اعلان اول، از مقدار پیش فرض استفاده شده، که در این حالت `jan = 0`، `feb = 1`، `mar = 2` و `apr = 3` خواهند بود. اما در اعلان دوم، برنامه‌نویس بنا به نیاز خود، تشخیص داده تا به هر یک از عناصر `enum` مقداری دلخواه را نسبت دهد.

هر چند به تمامی عناصر `enum` مقداری نسبت داده می‌شود ولی از این مقدار نمی‌توان بطور مسقیم در تخصیص دهی مقدار به متغیری دیگر استفاده نمود. به مثال زیر توجه نمایید :

```
int x = Months.jan; //این دستور نادرست است
int x = (int) Months.jan; //صحیح
```

برای استفاده از مقدار تخصیص داده شده به عناصر `enum`، باید از `Casting` استفاده نماییم. بدین معنا که باید نوع متغیری را که می‌خواهیم مقدار را به آن نسبت دهیم، باید مشخص شود. در مثال فوق `(int) Months.jan` معین می‌کند که مقدار

تخصیص داده شده به `jan` به متغیری نسبت داده می‌شود که از نوع `int` است و یا به عبارت صحیح تر، مقدار تخصیص داده شده به عنصر `enum`، در فرمت `int` به متغیر مورد نظر تخصیص داده می‌شود.

در ادامه مبحث، توجه شما را به مثالی دیگر درباره `enum` جلب می‌نمایم. توجه نمایید که نکات جدیدی نیز در این مثال گنجانده شده‌اند.

مثال ۲-۱۷: ساخت `enum` پایه و تخصیص دهی اعضای آن

```
using System;

// declares the enum
public enum Volume : byte
{
    Low = 1,
    Medium,
    High
}

class EnumBaseAndMembers
{
    static void Main()
    {
        // create and initialize
        // instance of enum type
        Volume myVolume = Volume.Low;

        // make decision based
        // on enum value
        switch (myVolume)
        {
            case Volume.Low:
                Console.WriteLine("The volume has been turned Down.");
                break;
            case Volume.Medium:
                Console.WriteLine("The volume is in the middle.");
                break;
            case Volume.High:
                Console.WriteLine("The volume has been turned up.");
                break;
        }
        Console.ReadLine();
    }
}
```

با توجه به مثال ۲-۱۷ با نحوه تغییر نوع پایه یک `enum` آشنا می‌شوید. همانطور که ملاحظه می‌نمایید، نوع پایه این `enum` به `byte` تغییر یافته است. این امر بیان میدارند که تنها مقادیری از نوع `byte` قابل تخصیص به عناصر `enum` هستند.

همانطور که قبلاً نیز اشاره شد، مقدار پیش فرض برای اولین عضو `enum` یعنی `Low` برابر با صفر است. چون در این مثال مقدار `Low` را برابر با یک قرار داده ایم، از اینرو مقادیر دو عضو دیگر آن نیز بصورت `Middle=2` و `High=3` تغییر خواهند یافت.

نکات پیشرفته درباره `enum`

در زبان `C#` هر نمونه از `enum` فضایی معادل با ۴ بایت از حافظه را اشغال می نمایند. این مطلب با استفاده از کلمه کلیدی `sizeof` قابل بررسی لسا. از اینرو از `enum` میتوان به عنوان یک ساختمان داده مناسب و کارا یاد کرد.

نکته بسیار مهم و جالب در مورد `enum` در زبان برنامه نویسی `C#` آنست که، برای هر کلاس `enum` موجود در کد برنامه، مقادیر رشته ای تخصیص داده شده به عناصر `enum` در یک اسمبلی و بصورت `Metadata` ذخیره می گردند، از اینرو دسترسی به این مقادیر رشته ای در کد میسر می شود. همچنین می توان از متدهای مختلف مرتبط با `enum` نیز استفاده نمود. به مثال ساده زیر توجه نمایید :

```
enum Language
{
    CSharp, MCpp, VBNet, JScript, IL
}
class App
{
    public static void Main()
    {
        Console.WriteLine("Write the number of the selected Language");
        string[] langAr = Enum.GetNames(Type.GetType("Language"));
        for(int i=0; i<langAr.Length; i++)
        {
            Console.WriteLine(i + "." + langAr[i]);
        }
        Language myLang=(Language)Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Your Language of choice is: " + myLang);
    }
}
```

Overload کردن عملگرها در C#

مطالبی که در این قسمت مورد بررسی قرار خواهند گرفت به شرح زیر می باشند :

Overload کردن عملگرها چیست ؟

درک اینکه چه زمانی از Overload کردن عملگرها استفاده می کنیم.

چگونگی Overload کردن عملگرها

قوانین معمول در Overload کردن عملگرها

در این مبحث می خواهیم درباره Overload کردن عملگرهای زبان C# صحبت کنیم. Overload کردن عملگرها بدین معناست که با استفاده از عملگرهای موجود در زبان C#، عمل دیگری بغیر از عمل در نظر گرفته شده برای آن عملگر را برای آن تعریف نماییم. در این مبحث با این مفهوم به طور کامل آشنا خواهید شد.

نگاهی بر Overload کردن عملگرها

همانطور که می دانید، در C# همانند سایر زبانهای برنامه سازی، عملگرهای متعددی وجود دارند (برای بررسی عملگرها می توانید به "عملگرها در C#" مراجعه نمایید). این عملگرها برای انواع از پیش تعریف شده در زبان C# مورد استفاده قرار می گیرند. اما در موارد مورد نیاز می توان برای انواع تعریف شده توسط کاربر نیز، عملگرهای خاص مورد نظر را تعریف نمود. با استفاده از Overload کردن عملگرها می توان از عملگرهای تعریف شده نیز همانند عملگرهای موجود در زبان استفاده کرد.

برای درک بهتر اهمیت استفاده از Overload کردن عملگرها، فرض کنید می خواهید عملیات ریاضی را بر روی ماتریس ها انجام دهید. برای انجام این کار، مثلاً ماتریسی دو بعدی ایجاد می کنید و از آن استفاده می کنید. اما می دانید که می خواهید از این کد تولید شده در برنامه های دیگر نیز استفاده کنید.

برای این منظور، یک نوع جدید با نام Matrix اعلان می کنید که این نوع جدید می تواند یک کلاس و یا یک struct باشد. (برای بررسی کلاسها به "کلاسها در C#" و برای بررسی ساختارها به "ساختارها در C#" رجوع نمایید). حال که نوع جدیدی با عنوان Matrix را اعلان نموده اید، مسلماً می خواهید از روی آن نمونه هایی تولید کرده و عملیات ریاضی نظیر جمع و ضرب را بر روی آنها اجرا نمایید. برای انجام چنین عملیاتی می توان دو متد Add() و Product() را پیاده سازی نمود و از آنها استفاده نمود. مثلاً شکل استفاده از این متدها بسته به نحوه آنها می تواند به یکی از شکلهای زیر باشد :

```
Matrix result = mat1.Add(mat2); // instance
```

```
Matrix result = Matrix.Add(mat1, mat2); // static
```

```
Matrix result = mat1.DotProduct(mat2).DotProduct(mat3); // and so on...
```

و یا هر حالت دیگری که می توانید برای آن در نظر بگیرید. تعریف این چنین متدهایی و استفاده از آنها، دشوار، غیر عادی و دشوار است. اما در صورتیکه بتوان برای جمع از + استفاده نمود، حالتی بسیار مناسب رخ می دهد. حال فرض کنید می توانیم از + به جای عمل جمع ماتریسها و از * برای ضرب ماتریسها استفاده کنیم. در اینصورت سه فرمت بالا به شکل زیر در خواهند آمد :

```
Matrix result = mat1 + mat2;
```

```
Matrix result = mat1 * mat2;
```

```
Matrix result = mat1 * mat2 * mat3 * mat4;
```

همانطور که ملاحظه می کنید، استفاده از چنین فرمتی به مراتب آسانتر از تعریف متدهایی برای اجرای عملیاتی این چنین است. همچنین استفاده از این فرمت جدید در مسایل پیچیده و دارای عملیات زیاد، بسیار ساده تر و مطمئن تر است.

موارد نامناسب استفاده از **Overload** کردن عملگرها

قبل از اینکه بخش پیاده سازی **Overload** کردن عملگرها بپردازیم لازم است تا بیان داریم که استفاده از **Overload** کردن عملگرها در همه موارد کارآیی ندارد و می تواند باعث گمراهی شود. بهترین موارد استفاده از **Overload** کردن عملگرها، مواردی هستند که عملگری که **Overload** می شود، واقعا بر روی نوع مورد نظر تعریف شده باشد و دارای مفهومی حداقل ریاضی باشد. برای مثال در نظر بگیرید پارکینگ داریم که می خواهیم ورود ماشین در آن را شبیه سازی نماییم. در این حالت **Overload** کردن عملگر + برای ورود ماشین به پارکینگ مناسب نیست، چراکه در این مورد عملگر + مفهومی نمی تواند داشته باشد. توجه نمایید که بیشتر از **Overload** کردن عملگرها در مواردی استفاده می شود که به نحوی به مسایل ریاضی مربوط هستند و یا عملگر مورد نظر برای شیء خاص تعریف شده بوده و یا استفاده از این عملگر بر روی شیء، ابهام ایجاد نکند.

عملگرهای **Overload** شونده

همانطور که تا کنون ملاحظه نموده اید، تمامی عملگرهای زبان **C#** دارای پیاده سازی داخلی هستند که می توان از این عملگرها در هر عبارتی استفاده نمود. اما مواردی نیز وجود داشتند که برای سهولت کار می توانیم عملگر خاصی را بطور مورد نظر خود پیاده سازی نماییم. پس از اینکه عملگری **Overload** شد، پیاده سازی انجام شده توسط کاربر بر پیاده سازی پیش فرض تقدم پیاده کرده و تنها در صورتیکه عملگر **Overload** شده دارای پیاده سازی نباشد، از پیاده سازی از پیش تعریف شده استفاده خواهد شد. همانطور که می دانید در **C#** دو نوع عملگر وجود دارد. عملگرهای یگانی (**Unary**) که قابل **Overload** کردن هستند به شرح زیر می باشند :

```
+ - ! ~ ++ -- true false
```

عملگرهای باینری قابل **Overload** شدن نیز به شرح زیر می باشند :

```
+ - * / % & | ^ << >> == != > < >= <=
```

توجه نمایید، هر چند عملگرهای **true** و **false** هیچگاه بطور صریح بعنوان عملگر در عبارات استفاده نمی شوند، اما آنها را نیز عملگر می خوانیم چراکه در بسیاری از عبارات منطقی و شرطی از آنها بعنوان عملگر استفاده می گردد.

توجه نمایید که تنها عملگرهای مشخص شده در بالا قابلیت Overload شدن را دارند و نمی‌توان سایر عملگرهای زبان C# را Overload نمود. همچنین توجه کنید که Overload کردن یک عملگر باعث Overload شدن ضمنی سایر عملگرهای مرتبط با آن نیز می‌شود. برای مثال Overload کردن عملگر + باعث Overload شدن ضمنی عملگر ترکیبی += نیز می‌شود. البته توجه نمایید که عملگر انتساب یا = هیچ‌گاه Overload نمی‌شود.

نکته دیگری که باید در مورد عملگرهای Overload شده در نظر گرفت آنست که Overload کردن عملگرها خواصی نظیر حق تقدم و یا شرکت‌پذیری عملگر را تغییر نمی‌دهد. بعنوان مثال عملگر /، عملگری باینری با حق تقدم مشخص و شرکت‌پذیری از چپ است.

پیاده‌سازی عملگر Overload شده

پیاده‌سازی یک عملگر Overload شده تقریباً شبیه به پیاده‌سازی متدی استاتیک است، با این تفاوت که در اینجا لازم است از کلمه کلیدی operator و عملگر مورد نظر استفاده نماییم. در زیر نمونه‌ای از ساختار کلی عملگر ضرب که برای ماتریسها در نظر گرفته بودیم را مشاهده می‌نمایید.

```
public static Matrix operator *(Matrix mat1, Matrix mat2)
{
    // dot product implementation
}
```

همانطور که مشاهده می‌کنید، متد استفاده شده حتماً باید بطور استاتیک تعریف گردد. از کلمه کلیدی operator نیز پس نوعی که می‌خواهیم برای آن عملگری را Overload نماییم، قرار می‌گیرد که در اینجا Matrix نوع مورد نظر ما است. پس از کلمه کلیدی operator، عملگری که می‌خواهیم Overload کنیم را قرار داده و سپس پارامترهایی که عملگر بر روی آنها اعمال می‌شوند را قرار می‌دهیم. در مثال ۱-۱۸، نمونه‌ای از Overload کردن عملگرها را مشاهده خواهید کرد.

مثال ۱-۱۸: نمونه‌ای از Overload کردن عملگرها

```
using System;

class Matrix3D
{
    public const int DIMSIZE = 3;
    private double[,] matrix = new double[DIMSIZE, DIMSIZE];

    // امکان تخصیص مقدار را برای فراخواننده فراهم می‌کند.
    public double this[int x, int y]
    {
        get { return matrix[x, y]; }
        set { matrix[x, y] = value; }
    }

    // Overload کردن عملگر + برای استفاده بر روی ماتریسها
```

```

public static Matrix3D operator +(Matrix3D mat1, Matrix3D mat2)
{
    Matrix3D newMatrix = new Matrix3D();

    for (int x=0; x < DIMSIZE; x++)
        for (int y=0; y < DIMSIZE; y++)
            newMatrix[x, y] = mat1[x, y] + mat2[x, y];

    return newMatrix;
}

class MatrixTest
{
    // در متد InitMatrix از آن استفاده می‌شود.
    public static Random rand = new Random();

    static void Main()
    {
        Matrix3D mat1 = new Matrix3D();
        Matrix3D mat2 = new Matrix3D();

        // ماتریسها با مقادیر تصادفی مقداردهی می‌شوند.
        InitMatrix(mat1);
        InitMatrix(mat2);

        // ماتریسها در خروجی نمایش داده می‌شوند.
        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);
        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);

        // عمل جمع ماتریسها صورت گرفته و نتیجه محاسبه می‌گردد.
        Matrix3D mat3 = mat1 + mat2;

        Console.WriteLine();
        Console.WriteLine("Matrix 1 + Matrix 2 = ");
        PrintMatrix(mat3);
    }

    // متدی که در آن ماتریسها با مقادیر تصادفی مقداردهی می‌شوند.
    public static void InitMatrix(Matrix3D mat)
    {
        for (int x=0; x < Matrix3D.DIMSIZE; x++)
            for (int y=0; y < Matrix3D.DIMSIZE; y++)
                mat[x, y] = rand.NextDouble();
    }

    // متد چاپ ماتریس در خروجی.

```

```

public static void PrintMatrix(Matrix3D mat)
{
    Console.WriteLine();
    for (int x=0; x < Matrix3D.DIMSIZE; x++)
    {
        Console.Write("[ ");
        for (int y=0; y < Matrix3D.DIMSIZE; y++)
        {
            // فرمت‌دهی خروجی.
            Console.Write("{0,8:#.000000}", mat[x, y]);

            if ((y+1 % 2) < 3)
                Console.Write(", ");
        }
        Console.WriteLine(" ]");
    }
    Console.WriteLine();
}
}

```

در مثال ۱-۱۸، عملگر + مورد Overload شدن قرار گرفته است. برای تمرکز بیشتر بر روی کد، قسمت مربوط به Overload شدن عملگر + را در زیر آورده‌ام:

```

public static Matrix3D operator +(Matrix3D mat1, Matrix3D mat2)
{
    Matrix3D newMatrix = new Matrix3D();

    for (int x=0; x < DIMSIZE; x++)
        for (int y=0; y < DIMSIZE; y++)
            newMatrix[x, y] = mat1[x, y] + mat2[x, y];

    return newMatrix;
}

```

عملگر همیشه بطور استاتیک اعلان می‌شود، چراکه متعلق به یک نوع کلی است و مربوط به نمونه‌ای خاص نمی‌باشد. نوع بازگشتی، Matrix3D است و تنها چیزی که این متد را از یک متد عادی متمایز می‌نماید استفاده از کلمه کلیدی operator و عملگر + است. پیاده‌سازی عملگر Overload شده باعث ایجاد نمونه‌ای جدید از Matrix3D شده و عمل جمع ماتریس را انجام می‌دهد.

نکاتی چند در مورد Overload کردن عملگرها

زبان C# قوانینی برای Overload کردن عملگرها اعمال می‌کند. یکی از این قوانین آنست که عملگر Overload شده باید از نوعی که مورد استفاده قرار می‌گیرد، اعلان شود.

قانون بعدی اینست که به هنگام پیاده‌سازی عملگرهای مقایسه‌ای نظیر <، > و == باید حالت‌های ترکیبی آنها را نیز پیاده‌سازی نمود. برای مثال در صورتیکه عملگر > را Overload می‌کنید، باید عملگر >= را پیاده‌سازی نمایید.

نکته دیگر اینکه، پس از Overload کردن عملگرها، عملگرهای ترکیبی آنها نیز قابل استفاده هستند. توجه نمایید که عملگرهای مقایسه‌ای که در بالا اشاره شد از این قاعده مستثنا هستند. همانطور که در قبل اشاره شد، پیاده‌سازی عملگر + باعث می‌شود تا بتوان از += نیز استفاده نمود.

ساختمان داده‌ها در C# (با رویکردی بر C# 2.0)

این مطلب، آغاز یک سری شش قسمتی از مطالب مهم و کلیدی درباره ساختمان داده و کاربر آن در توسعه و طراحی نرم افزار است. در قسمت اول، مختصری درباره ساختمان داده‌ها، تعریف ساختمان داده، آنالیز کارایی یک ساختمان داده و بررسی اهمیت این آنالیز صحبت خواهیم نمود. در این بخش همچنین درباره دو ساختمان داده بسیار مهم و رایج در .Net Framework، یعنی آرایه‌ها و لیست‌ها صحبت خواهیم کرد.

توجه : مطالب موجود در این ۶ قسمت، منطبق بر ساختار .Net Framework 2.0 و C# 2.0 تنظیم شده‌اند.

پیشنیاز این مطالب، آشنایی با زبان برنامه‌سازی C# است. برای فراگیری این زبان می‌توانید به لینک زیر مراجعه نمایید :

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

قسمت اول : مقدمه‌ای بر ساختمان داده

مطالب مورد بحث در این قسمت بشرح زیر می‌باشند :

- مقدمه
- آنالیز کارایی یک ساختمان داده
- ساختمان داده خطی، همگون و بسیار متداول : آرایه (Array)
- ایجاد ساختمان داده‌های کارآ، قابل استفاده مجدد و Type-Safe

- لیست (List) : ساختمان داده ای همگون

- نتیجه و جمعه بندی

مقدمه

در طول این مطلب، با چندین ساختمان داده موجود در **Net Framework Base Class**، و همچنین ساختمان داده هایی که توسط کاربر ایجاد می شوند، آشنا خواهید شد. اگر با مفهوم ساختمان داده آشنایی ندارید به تعریف زیر توجه نمایید :

تعریف : ساختمان داده، کلاسی است که جهت سازماندهی داده ها مورد استفاده قرار می گیرد و از عملیات مختلف قابل اجرا بر روی این داده ها، پشتیبانی می نماید.

معمول ترین و آشنا ترین ساختمان داده، آرایه است که شامل مجموعه ای از داده ها است که پشت سر هم قرار گرفته اند و از طریق یک اندیس مشخص قابل دسترسی هستند.

قبل از شروع این قسمت، مطالب مورد بررسی در مجموعه شش قسمتی مربوط به ساختمان داده را مرور می کنیم تا هدف نهایی از این سری مطالب برای شما مشخص گردد.

در قسمت اول، نگاهی بر اهمیت ساختمان داده ها و چگونگی تاثیر آنها بر کارایی یک الگوریتم خواهیم کرد. برای تعیین تاثیر کارایی یک ساختمان داده بر روی یک الگوریتم، لازم است تا کلیه عملیات موجود در ساختمان داده به دقت مورد بررسی قرار گیرند. در ابتدا تمرکز خود را بر روی دو ساختمان داده آرایه و لیست جمع می کنیم، چراکه با این دو ساختمان داده از قبل آشنایی داشته و می توانیم راحت تر به آنالیز آنها بپردازیم. برای آنالیز کارایی آنها، به بررسی عملیات موجود در هر یک از آنها و کارایی هر یک از این عملیات خواهیم پرداخت.

در قسمت دوم، به بررسی صف (Queue) و پشته (Stack) خواهیم پرداخت. همانند لیست ها، صف و پشته نیز مجموعه ای از داده ها هستند و این دو ساختمان داده در **Net Framework Base Class Library** وجود دارند. برخلاف لیست، که اعضای آن به هر ترتیبی قابل دسترسی هستند، دسترسی به اعضای صف و پشته تنها از طریق ترتیبی خاص قابل دسترسی است. سپس به بررسی چند برنامه و طریقه پیاده سازی صف و پشته خواهیم پرداخت و پس از آن **Hashtable** ها را مورد بررسی قرار می دهیم. **Hashtable** ها امکان دسترسی به عناصر را بصورت مستقیم فراهم می نمایند (همانند آرایه یا **ArrayList** ها) ولی اندیس مورد استفاده در آن کلیدهایی رشته ای هستند. (**String Keys**)

هر چند ساختمان داده هایی مانند آرایه و لیست، برای دسترسی مستقیم به داده ها در زمانیکه با حجم زیادی از داده ها سر و کار داریم بسیار مفید هستند، این ساختمان داده ها برای جستجو درون داده ها، از بهینگی کمتری برخوردارند. در قسمت سوم، به بررسی ساختمان داده درخت جستجوی دودویی (**Binary Search Tree**) می پردازیم که روشی مناسب و بهینه برای جستجوی داده های موجود در یک مجموعه است.

اگرچه استفاده از درخت جستجوی دودویی باعث کاهش زمان جستجو می شود، اما دارای نواقص و کاستی هایی نیز می باشد. در قسمت چهارم، به بررسی **SkiList** ها می پردازیم که ترکیبی از درختهای دودویی و لیست های پیوندی (**Linked List**) است.

در قسمت پنجم، به بررسی ساختمان داده هایی پرداخته می شوند که می توانند نشان دهنده گراف (**Graph**) باشند. یک گراف مجموعه ای از گره ها (**Nodes**) است که هر یک از آنها با لبه هایی (**Edge**) به گره های دیگر متصل شده اند. برای مثال، نقشه شهرها را میتوان با یک گراف پیاده سازی کرد که در آن شهرها همان گره ها و راه ها و اتوبانهای بین شهرها نشان دهنده **Edge** ها می باشند. بسیاری از مسایل دنیای واقعی با استفاده از مفهوم گرافها بطور انتزاعی قابل پیاده سازی هستند.

نهایتاً در قسمت ششم، به بررسی ساختمان داده هایی می پردازیم که نشان دهنده **Sets** و **Disjoined Sets** هستند. **Set**، مجموعه ای است از داده ها که بدون هیچ گونه ترتیبی در کنار یکدیگر قرار گرفته اند. **Disjoined Set**، مجموعه ای از **Set** هاست که هیچ عنصر مشترکی با یکدیگر ندارند. این دو مفهوم در پیاده سازی برنامه های امروزه کارایی زیادی دارند و در قسمت ششم به بررسی و نحوه استفاده از آنها خواهیم پرداخت.

آنالیز کارایی ساختمان های داده

هنگامیکه با یک مسئله برنامه نویسی مواجه می شویم، اغلب برنامه نویسان و طراحان نرم افزار به دنبال پیدا کردن و طراحی الگوریتمی هستند تا بتوانند بواسطه آن کارآیی برنامه خود را افزایش داده و نیازهای کاربر را به بهترین شکل ممکن برآورده نمایند. از دید یک کاربر نرم افزار نیز مفید بودن برنامه اهمیت دارد و کمتر کسی پیدا می شود که به الگوریتم پشت پرده مورد استفاده در برنامه اهمیت دهد. اما استفاده از یک الگوریتم قوی و کارآ در زمینه برنامه یا نرم افزاری که تولید می شود، می تواند نقش بسیار مهمی در افزایش کارآیی برنامه داشته باشد. بعنوان مثال، جستجو درون یک آرایه را در نظر بگیرید. با استفاده یک الگوریتم جستجوی ساده و عادی، به تعداد اعضای موجود در آرایه باید عمل جستجو انجام گیرد. با استفاده از درخت جستجوی دودویی یا SkipList ها، مدت زمان جستجو برای یک عنصر نسبتی لگاریتمی با تعداد اعضای موجود در آرایه پیدا می کند و باعث کاهش زمان جستجو می شود. زمانیکه در انبوهی از داده های بخواهیم به جستجو بپردازیم، نوع الگوریتم و ساختمان داده مورد استفاده بسیار می تواند در زمان انجام جستجو موثر باشد، بطوری که می توان اختلاف آنرا بر حسب ثانیه و حتی دقیقه حس نمود.

اگرچه ساختمان داده مورد استفاده در یک الگوریتم به شدت می تواند کارآیی آن را تحت تاثیر قرار دهد، اما روشهای دقیق و مناسبی نیز برای مقایسه و بدست آوردن این کارآیی باید وجود داشته باشند. مسئله ای که برای ما بعنوان یک طراح نرم افزار یا برنامه نویس می تواند اهمیت داشته باشد، کارآیی ساختمان های داده مختلف به هنگام افزایش حجم داده ها است. و این بدان معناست که برای هر عنصر جدیدی که وارد ساختمان داده ما می شود، زمان کار الگوریتم ما به ه شکلی تحت تاثیر قرار می گیرد.

به مثال زیر توجه کنید، فرض کنید باید برنامه ای بنویسید که آرایه ای از رشته ها را بعنوان ورودی دریافت نماید که این آرایه حاوی نام تعدادی فایل است. وظیفه برنامه شما اینست که درون این آرایه به دنبال فایلی با پسوند خاصی بگردد. نمونه ساده این برنامه می تواند بشک زیر باشد :

```
public bool DoesExtensionExist(string [] fileNames, string extension)
{
    int i = 0;
    for (i = 0; i < fileNames.Length; i++)
        if (String.Compare(Path.GetExtension(fileNames[i]), extension, true) == 0)
            return true;
```

```
return false; // If we reach here, we didn't find the extension
}
}
```

به نگاهی دقیق تر به این برنامه، متوجه می شوید که در بدترین حالت، یعنی زمانیکه فایل مورد نظر در آرایه وجود نداشته باشد و یا وجود داشته باشد اما در آخرین خانه آرایه جای داشته باشد، باید تمامی عناصر آرایه را یکبار مورد بررسی و جستجو قرار دهیم. برای آنالیز مسائل، بطور مثال برای آنالیز مرتب سازی عناصر آرایه (Sort)، به شکل زیر باید فکر کنیم: "فرص می کنم آرایه ای با n عنصر دارم، اگر عنصر دیگری به این آرایه اضافه کنم $n+1$ عنصر خواهم داشت، در این حالت زمان اجرای برنامه من چقدر خواهد شد؟". توجه نمایید که کلمه "زمان اجرا" یا **Running Time** عملاً به معنا محاسبه دقیق زمان اجرا برنامه نیست، بلکه محاسبه تعداد مراحل و زمان اجرای این مراحل برای یک عمل خواسته شده است. برای مثال در مورد آرایه، تعداد مراحل به تعداد دفعات دسترسی به عناصر آرایه است. برای جستجو درون یک آرایه، در صورتیکه $n+1$ عنصر در آرایه داشته باشیم، باید $n+1$ بار مقدار مورد نظر خود را با عناصر آرایه مقایسه نماییم. از اینرو می گوئیم مدت زمان جستجو درون یک آرایه بطور **خطی** با تعداد عناصر موجود در آرایه رابطه **مستقیم** دارد.

این روش آنالیز که در اینجا مطرح گردید، به روش آنالیز آسیمپتوتیک (**Asymptotic Analyze**) معروف است. روش علامتگذاری که در این روش آنالیز مورد استفاده قرار می گیرد، به روش نشانه گذاری **O (Big-Oh Notation)** معروف است. برای نشان دادن کارایی جستجو در آرایه غیر مرتب با استفاده از این روش نشانه گذاری، بشکل زیر عمل می شود $O(n)$ که نشان دهنده رابطه زمان با تعداد اعضا است. بدین ترتیب با استفاده از $O(n)$ برای نشان دادن کارایی یک ساختمان داده، این مفهوم بدست می آید که رابطه ای خطی و مستقیم بین افزایش تعداد اعضای n به $n+1$ و زمان انجام عملیات وجود دارد. بدین ترتیب علامت **O** نشان دهنده روش نشانه گذاری و روش آنالیز کارایی است و n نشان دهنده چگونگی افزایش مراحل پردازش عمل بسته به تعداد عناصری است که می خواهند مورد پردازش قرار گیرند.

برای محاسبه زمان اجرای یک قطعه کد با استفاده از **O-Notation** یا همان **Asymptotic Analyze** می توان از چند مرحله ساده زیر استفاده نمود:

- ۱- بدست آوردن مراحل که زمان اجرای یک الگوریتم را شکل می دهند. همانطور که قبلاً ذکر شد، این مراحل برای یک آرایه، خواندن و نوشتن یا همان دسترسی به عناصر آرایه است و این مراحل برای ساختمان های داده ای دیگر متفاوت خواهد بود. باید توجه داشته باشید که برای آنالیز کارایی یک ساختمان داده، تنها به مراحل که توسط خود ساختمان داده ی مورد نظر انجام می شود توجه می گردد و عملیات ساده ای که توسط کامپیوتر انجام می شوند را در این محاسبه

به حساب نمی آوریم. برای مثال در کد بالا، برای محاسبه کارایی تنها به تعداد دفعات لازم برای دسترسی به عناصر آرایه توجه کرده و به زمان های لازم جهت ایجاد متغیرها و با زمان محاسبه تساوی دو رشته اهمیت نداده ایم.

۲- خطهایی از برنامه که در الگوریتم شما تاثیر دارند و می خواهید کارایی را برای آنها محاسبه کنید را پیدا کرده و در کنار آنها عدد ۱ قرار دهید.

۳- اگر این خطوط که آنها را با عدد ۱ مشخص کرده اید خود یک حلقه تکرار هستند، بجای عدد ۱ در کنار آنها عدد معادل بیشترین تعداد تکرار حلقه را قرار دهید. اگر در جایی حلقه تو در تو دارید، باید از ضرب بیشترین تعداد دفعات تکرار حلقه ها استفاده کنید.

۴- بزرگترین عدد نوشته شده را پیدا کنید. این عدد زمان اجراست.

حال برای روشن تر شدن موضوع همین مراحل را برای کد نوشته شده در بالا اجرا میکنیم. در کد نوشته شده در بالا، مراحل مورد نظر برای ما، تعداد دفعات دسترسی به عناصر آرایه است. با توجه به کد و با توجه به مرحله ۲، تنها در یک خط کد، دسترسی به عناصر آرایه وجود دارد، پس عدد ۱ را در کنار این خط قرار می دهیم. با توجه به مرحله ۳، این خط در یک حلقه تکرار قرار دارد و این حلقه n با تکرار می شود که n تعداد عناصر آرایه است. طبق مرحله ۳، عدد ۱ را پاک کرده و بجای آن n در کنار این خط قرار می دهیم. N بزرگترین مقدار نوشته شده است از اینرو زمان اجرا $O(n)$ است.

$O(n)$ یکی از صدها زمان اجرای موجود در آنالیز Asymptotic است. برخی دیگر از زمانهای اجرا عبارتند از $O(\log_2 n)$, $O(n \log_2 n)$, $O(n^2)$, $O(2^n)$ و بدون اینکه وارد جزئیات پیچیدگیهای ریاضیاتی Notation_O شویم، بیان میداریم که هرچه مقدار داخل پرانتز در این آنالیز کوچکتر باشد، عملیات بر روی ساختمان داده کارآمدتر است. برای مثال ساختمان داده ای که زمان اجرای آن $O(\log n)$ است به مراتب کارآمدتر از ساختمان داده ایست که زمان اجرای آن $O(n)$ است. چراکه $\log n < n$

نکته: برای یادآوری باید بگویم که $\log_a b = y$ روش دیگری برای نمایش $a^y = b$ است. از اینرو $\log_2 n$ رشد کندتری نسبت به n دارد، چراکه اگر $n=8$ فرض کنیم، $\log_2 n = 3$ می شود در حالیکه $n=8$ است. در قسمت سوم از این سری مطالب، با درخت های جستجوی دودویی آشنا می شویم که زمان اجرایی آنها $O(\log_2 n)$ است.

زمان اجرای Asymptotic و الگوریتمهای دنیای واقعی

زمان اجرای **Asymptotic** روشی برای محاسبه کارایی الگوریتم در زمانی است که تعداد عناصر مورد نظر آن به سمت بینهایت میل می کند. هنگامیکه گفته می شود زمان اجرایی یک الگوریتم از الگوریتم دیگر بیشتر است، از لحاظ ریاضی بدین معناست که مراحل اجرایی و زمان اجرایی این مراحل در الگوریتم زمان بر بیشتر از الگوریتمی است که زمان اجرایی کمتری دارد. اما باید توجه کرد که در مواردی، الگوریتم هایی با مراحل اجرایی کم، با اینکه دارای زمان اجرایی بالایی هستند، از الگوریتم هایی که زمان اجرایی پائین تری دارند، سریعتر اجرا می شوند. همانطور که گفته شد این امر معمولاً در مواردی صادق است که الگوریتمی که زمان اجرایی آن بالاست، دارای مراحل بسیار کمی باشد.

بعنوان مثال، الگوریتم های بسیار زیاد و متفاوتی برای مرتب سازی آرایه ها وجود دارد و هر یک از آنها دارای زمان اجرایی متفاوتی هستند. یکی از ساده ترین و بدیهی ترین الگوریتم های مرتب سازی (Sort) الگوریتم مرتب سازی حبابی یا **Bubble Sort** است. زمان اجرایی این الگوریتم $O(n^2)$ است که این مقدار خاطر وجود دو حلقه تودرتو در این الگوریتم بوجود آمده است. در مقابل الگوریتم دیگری به نام **Merge Sort** وجود دارد که زمان اجرایی آن $O(n \log_2 n)$ است. زمان اجرایی **Merge Sort** به مراتب کمتر از زمان اجرایی **Bubble Sort** است. اما برای آرایه هایی با تعداد عناصر کم، زمان اجرایی **Bubble Sort** کمتر از زمان اجرای **Merge Sort** است! از آنجائیکه **Merge Sort** از یک ساختار بازگشتی استفاده میکند و نیز نیم آرایه **Sort** شده را در انتها با یکدیگر ادغام می نماید، در مورد آرایه های کوچک کارایی چندانی ندارد و چون **Bubble Sort** تنها از جابجا کردن عناصر مجاور آرایه استفاده می کند، از اینرو برای آرایه ها کوچک الگوریتمی مناسب تر است. (لطفاً در صورتیکه در مورد الگوریتم های **Sort** سوال و یا مشکلی دارید حتماً به من ایمیل بزنید) نهایتاً، مراحل کاری **Merge Sort** از **Bubble Sort** کمتر است اما از آنجائیکه وزن این مراحل بالاست (هر مرحله احتیاج به زمان پردازشی بالایی دارد) از اینرو برای آرایه هایی عظیم مفید است. حال آنکه برای آرایه هایی با حجم کم الگوریتم **Bubble Sort** مناسب تر است. پس علاوه بر توجه به زمان اجرایی یک الگوریتم، توجه به موارد استفاده و حجم داده مورد نظر نیز موثر است.

آرایه : ساختمان داده ای خطی، همگون، با امکان دسترسی مستقیم

آرایه ها یکی از ساده ترین و پر استفاده ترین ساختمانهای داده هستند که در تمامی زبانهای برنامه سازی دارای ویژگیهای مشترکی هستند :

- محتویات آرایه در یک فضای حافظه پیوسته ذخیره می شود
- تمامی عناصر یک آرایه باید از یک نوع یا از یک نوع مشتق شده باشند. از اینرو به آرایه ، ساختمان داده ای همگون گفته میشود.

- دسترسی به عناصر آرایه بصورت مستقیم و از طریق اندیس است.

عملیات متدوالی که بر روی آرایه ها انجام میشود نیز بشرح زیر است :

- تخصیص

- دسترسی

در زبان C# هنگامیکه آرایه ای ایجاد میشود، مقدار پیش فرض تهی (Null) برای آن در نظر گرفته میشود. برای مثال خط زیر آرایه تهی BoolArray را ایجاد میکند :

```
bool[] BoolArray;
```

قبل از اینکه بتوانیم از آرایه ها استفاده کنیم، می بایست نمونه ای از آن را ایجاد نماییم تا بتوانیم عناصر مورد نظر را در آن ذخیره کنیم :

```
arrayName = new arrayType[allocationSize];
```

این خط از کد، باعث تخصیص فضای پیوسته ای از حافظه در CLR-Managed Heap به اندازه تعداد allocationSize از arrayType میشود. (برای مثال اگر فضای مورد نیاز برای ذخیره سازی متغیری از نوع int برابر ۲ بایت باشد، برای آرایه ای که allocationSize آن ۵ است، به ۱۰ بایت حافظه نیاز است.) اگر arrayType از انواع مقداری (Value Type) باشد، آنگاه مقادیری از نوع arrayType و به تعداد allocationSize ایجاد میشود. (این مقادیر ایجاد شده unboxed هستند.) در صورتیکه arrayType از انواع مرجعی (Reference Type) باشند، آنگاه مرجع هایی از نوع arrayType و به تعداد allocationSize ایجاد میگردند. (در صورتیکه تفاوت بین انواع مرجعی و مقداری و فرق بین پشته و heap را نمیدانید، به درک انواع رایج در .Net مراجعه نمایید.)

برای درک چگونگی نگهداری عناصر آرایه توسط Net Framework، به مثال زیر توجه نمایید :

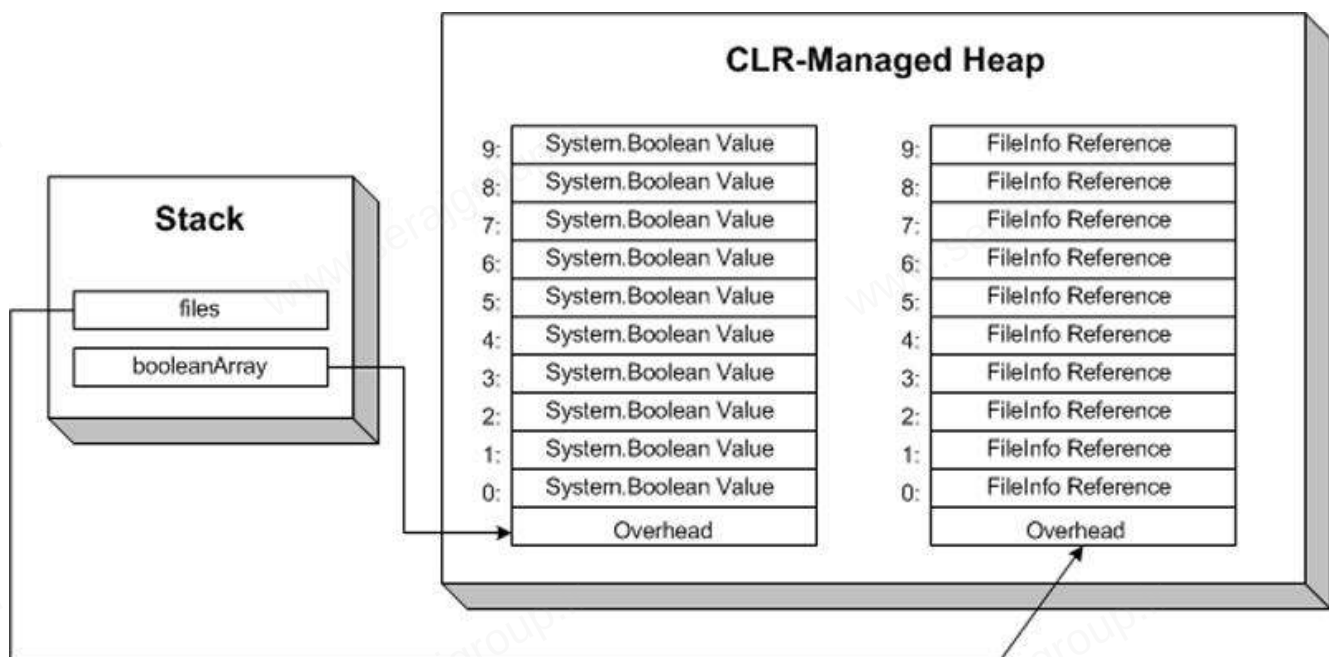

```
bool [] booleanArray;
```

```
FileInfo [] files;
```

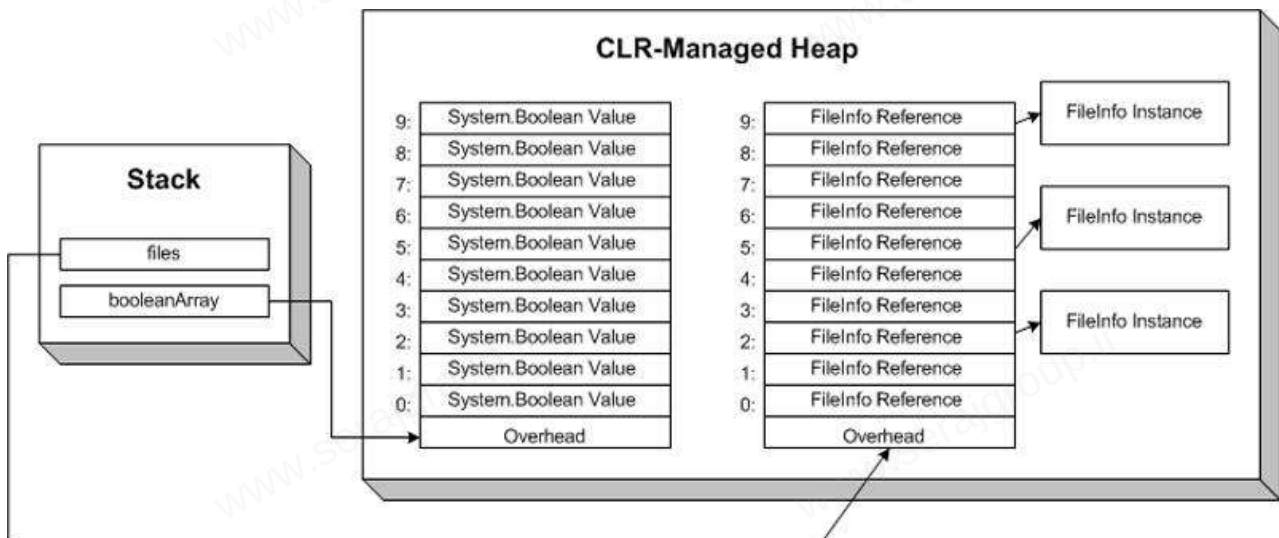
```
booleanArray = new bool[10];
```

```
files = new FileInfo[10];
```

در اینجا، آرایه `booleanArray` از نوع `System.Boolean` است در حالیکه آرایه `files` از نوع مرجعی `System.IO.FileInfo` میباشد. شکل زیر وضعیت `CLR-Managed Heap` را پس از اجرای این چند کد به نمایش در می آورد:



باید به این نکته نیز توجه نمایید که ۱۰ عنصر موجود در آرایه `Files`. مرجع هایی به نمونه هایی از `FileInfo` هستند.



در **.Net**، تمامی عناصر آرایه امکان نوشته شدن و نوشتن بر آرایه را دارند. نحوه دسترسی به عناصر آرایه به فرم زیر است :

```
// Read an array element
bool b = booleanArray[7];

// Write to an array element
booleanArray[0] = false;
```

زمان اجرایی دسترسی به یک آرایه را بصورت $O(1)$ نشان می دهیم، زیرا زمانی ثابت است. این به معنی آنست که، هر چقدر هم که عناصر آرایه زیاد باشند، زمان دسترسی به یک عنصر یکسان و ثابت است. علت ثابت بودن این زمان آنست که، عناصر آرایه بصورت متوالی در خانه های حافظه قرار می گیرند و به هنگام دسترسی به یک عنصر تنها کافی است تا نقطه شروع آرایه، طول عناصر آرایه و اندیس عنصر مورد نظر را داشته باشیم. در این حالت دسترسی به هر یک از عناصر آرایه زمانی واحد و ثابت را نیاز دارد.

نکته قابل توجه دیگر آنست که، در کدهای مدیریت شده، زمان دسترسی به عناصر آرایه اندکی با آنچه گفته شد متفاوت است. برای مثال در **.Net**، با هر درخواست برای دسترسی به عنصر یک آرایه، **CLR** اندیس مورد نظر را نیز چک میکند تا این اندیس درون آرایه قرار داشته باشد. در صورتیکه اندیس مورد نظر خارج از آرایه باشد، استثنای **IndexOutOfRangeException**

ایجاد میشود. (برای مثال اگر آرایه ای با ۱۰ عنصر داشته باشیم و اندیس دسترسی به عنصر آرایه را ۱۵ در نظر بگیریم، این استثناء رخ خواهد داد، چراکه اندیس مورد نظر خارج از مرزهای (Bound) آرایه است.) این کنترل باعث می شود تا به هنگام استفاده از آرایه ها مطمئن باشیم که به اشتباه وارد قسمت دیگری از حافظه نشده ایم. (برای مثال در برخی از نسخه های کامپایلر C این کنترل صورت نمی گیرد و شما می توانید به عنصر یازدهم یک آرایه ده عنصری دسترسی پیدا کنید !!! در این حالت اتفاقی که رخ می دهد آنست که، کامپایلر به اشتباه محتویات خانه ای از حافظه را به شما نشان میدهد که اصلا جزو آرایه مورد نظر نیست. این امر باعث بروز اشکالات فراوانی برای برنامه نویس به هنگام تست برنامه میشود. چراکه محتویات خانه ای از حافظه که به اشتباه اندیس دهی شده است، ممکن است تاثیرات نامطلوب و غیر قابل پیش بینی بر روی نتیجه برنامه داشته باشد و بررسی و کنترل منبع این خطا بسیار دشوار است.) توجه نمایید که انجام این کنترل باعث تغییر زمان اجرایی دسترسی به عناصر آرایه $O(1)$ نمیشود چراکه با افزایش تعداد عناصر آرایه این زمان تغییر نمیکند.

نکته : انجام چنین کنترلی بر روی اندیس دسترسی به عناصر آرایه، باعث افزایش کارایی در برنامه هایی میشود که بطور عظیمی با آرایه ها و دسترسی به عناصر آنها سروکار دارند. در کدهای مدیریت نشده (Unmanaged Code) امکان غیر فعال کردن این کنترل بر روی اندیس آرایه وجود دارد. برای مطالعه بیشتر در این زمینه میتوانید به فصل چهاردهم از کتاب "*Applied Microsoft .Net Framework Programming*" نوشته Jeffrey Richter مراجعه نمایید.

بهنگام کار با آرایه ها ممکن است نیاز به افزایش تعداد عناصر آرایه داشته باشیم. برای انجام چنین کاری لازم است تا نمونه جدیدی از آرایه مورد نظر را با طول جدید ایجاد کرده و عناصر آرایه فعلی را در آرایه جدید کپی نمایید. نمونه ای از کد مورد نظر برای اجرای چنین عملی در زیر آمده است :

```
// Create an integer array with three elements
int [] fib = new int[3];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;

// Redimension message to a 10 element array
int [] temp = new int[10];

// Copy the fib array to temp
```

```
fib.CopyTo(temp, 0);
```

```
// Assign temp to fib
```

```
fib = temp;
```

پس از اجرای آخرین خط در کد بالا، آرایه **fib** به یک آرایه ده عنصری اشاره میکند. عناصر ۳ تا ۹ این آرایه جدید مقدار پیش فرض **Int32** یعنی صفر را دارا خواهند بود.

آرایه ها یکی از بهترین ساختمان های داده برای نگهداری مجموعه ای از داده های همگون است که باید بطور مستقیم مورد دسترسی قرار گیرند. جستجو درون یک آرایه نا مرتب، زمان اجرای خطی دارد. توجه نمایید، در صورتیکه آرایه ای مرتب شده را مورد جستجو قرار دهیم، زمان اجرای جستجو به $O(\log n)$ تغییر پیدا خواهد کرد که زمانی نزدیک به زمان اجرایی درخت جستجوی دودویی است. توجه نمایید که در **.Net**، کلاس **Array** دارای متد **BinarySearch()** است که با استفاده از آن میتوان به جستجوی باینری درون آرایه مرتب شده پرداخت.

نکته : همانطور که میدانید **.Net Framework**، از آرایه های چند بعدی نیز پشتیبانی مینماید. از آنجائیکه زمان اجرای جستجو درون آرایه تک بعدی برابر با $O(n)$ است، زمان اجرای جستجو در یک آرایه k بعدی برابر با $O(n^k)$ خواهد بود. توجه کنید که در اینجا n تعداد عناصر آرایه در هر بعد است و منظور از n تعداد کل عناصر موجود در آرایه نیست.

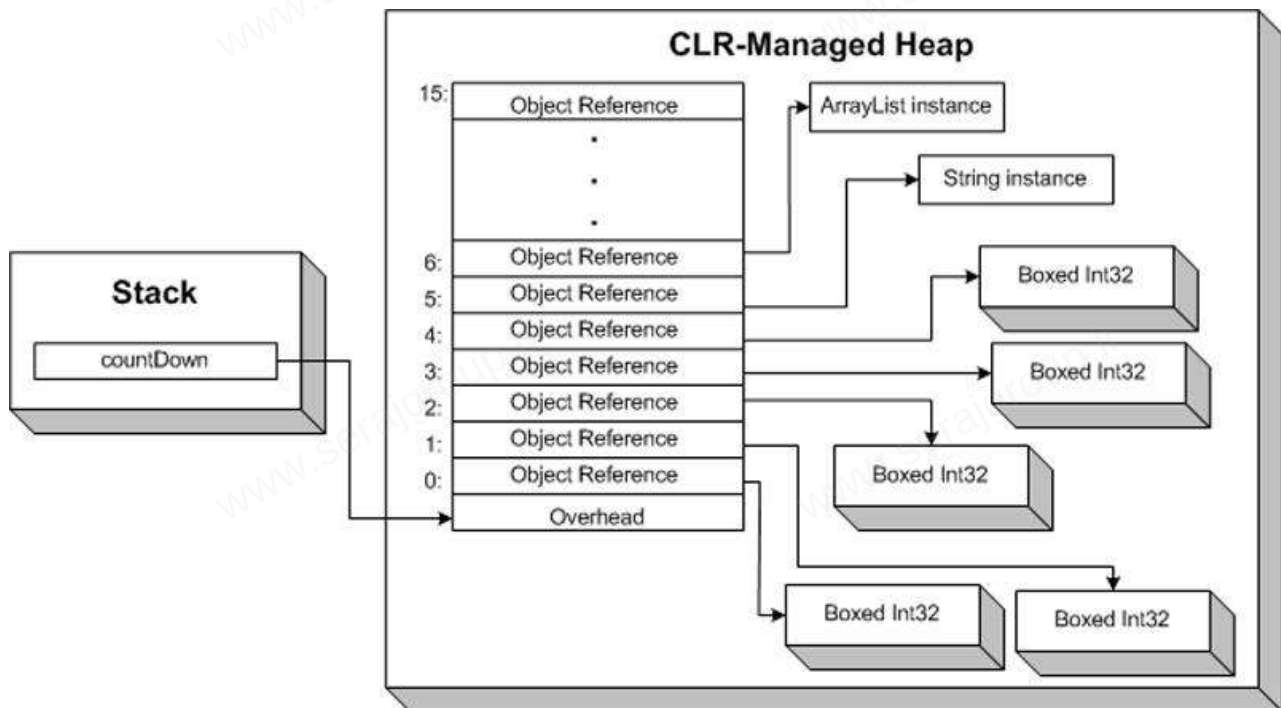
ساخت ساختمان داده ای مطمئن، کار آ و قابل استفاده مجدد

هنگامیکه میخواهیم ساختمان داده ای را برای مسئله ای خاص ایجاد نمائیم، ویژگیهای داخلی ساختمان داده در اغلب موارد، امکان شخصی سازی برای منطبق شدن با مسئله ما را فراهم مینماید. برای مثال فرض کنید میخواهید برنامه ای برای محاسبه حقوق کارکنان خود ایجاد نمایید. در این برنامه یکی از ساختمان داده های مورد نیاز میتواند آرایه **Employee** باشد. در عین حال شما نمی خواهید برای ورود هر کارمند جدید به شرکت، طول آرایه را تغییر دهید. از اینرو می توانید ساختمان داده ای شخصی تولید کنید که بصورت داخلی از یک آرایه استفاده می کند و کارایی آن را افزایش میدهد. برای مثال میتوان به تواناییهای آرایه، قابلیت افزایش حجم و جستجو را اضافه کرد.

ایجاد چنین ساختمان داده ای آنچنان می تواند مفید باشد که ممکن بخواهید از آن در برنامه های دیگر خود نیز استفاده نمایید. اما این ساختمان داده برای استفاده در برنامه های دیگر مفید نیست چراکه تنها عناصری از نوع `Employee` را میپذیرد. یک روش برای انعطاف پذیر کردن ساختمان داده مورد نظر آنست که بجای استفاده از آرایه ای از نوع `Employee`، آرایه ای از نوع `Object` را درون ساختمان داده خود استفاده کنیم. چون تمام انواع در `Net Framework`، از نوع `Object` مشتق میشوند، از اینرو آرایه ما میتواند هر نوع خاصی را بپذیرد. با این روش میتوانیم از ساختمان داده ایجاد شده در برنامه های دیگر نیز استفاده کنیم.

در `Net Framework`، چنین ساختمان داده ای بطور از پیش تعریف شده وجود دارد: کلاس `ArrayList` در `System.Collections.ArrayList` درون خود از آرایه ای از نوع `Object` استفاده کرده و امکان تغییر سایز خودکار را با افزایش تعداد عناصر مورد نیاز، فراهم مینماید. به دلیل اینکه `ArrayList` از آرایه ای از نوع `Object` استفاده می کند، از اینرو برنامه نویسان میتوانند از آن برای کار با انواع مختلف داده ای، بهره گیرند.

ایجاد انعطاف پذیری بیشتر، هزینه هایی دارد و در مورد `ArrayList` این هزینه مربوط به کارایی است. از آنجائیکه `ArrayList` از نوع `Object` استفاده میکند، بهنگام خواندن مقادیر از `ArrayList` میبایست بطور صریح عمل `casting` را با توجه به نوع داده ذخیره شده انجام داد. (`Casting` عملی است که در آن نوع داده مورد استفاده را مشخص میکنیم. برای مثال `x = (int)` در اینجا معین کرده ایم که متغیر `y` با نوع `int` به متغیر `x` تخصیص داده شود.) همانطور که به یاد دارید، گفتیم که آرایه هایی از نوع مقداری (همانند `System.Int32`، `System.Double` و `System.Boolean`) بصورت پیوسته در `Managed Heap` در فرم `Unboxed` خود ذخیره میشوند. اما آرایه مورد استفاده در `ArrayList`، از نوع `Object` است. از اینرو هنگامیکه `ArrayList` مورد نظر از انواع مقداری استفاده نماید، هر عنصر `ArrayList` مرجعی به نوع مقداری در فرم `Boxed` خود خواهد بود. این مطلب در شکل زیر به نمایش گذاشته شده است.



Boxing و Unboxing، هنگامیکه از `ArrayList` هایی با انواع مقداری استفاده می کنید و حجم دسترسی به عناصر `ArrayList` بالاست، میتوان بشكل محسوسی در کارایی برنامه تاثیر گذارد. همانطور که در شکل فوق نیز مشاهده میکنید، برای انواع مرجعی، فرمت شکل گیری حافظه در آرایه و `ArrayList` یکسان است.

استفاده از `ArrayList` ها، همچنین میتواند باعث بروز خطاهایی شود که تا زمان اجرای برنامه قابل لمس نخواهند بود. از آنجائیکه آرایه داخلی `ArrayList` از نوع `Object` است، قرار دادن نوع ناخواسته در `ArrayList` از سوی برنامه قابل قبول بوده و این مسئله باعث ایجاد خطا در برنامه میشود. همچنین این مسئله میتواند باعث پنهان شدن خطا تا زمان اجرا، تست و حتی، در بدترین شرایط، زمان تحویل نهایی برنامه شود.

Generic ها به کمک می آیند !

مسئله نوع و کارایی در `ArrayList` ها، با استفاده از `Generic` ها در `.Net Framework 2.0` بهبود یافته است. `Generic` این امکان را به طراح نرم افزار میدهد تا بهنگام ایجاد ساختمان داده انواع خاصی را برای استفاده در آن انتخاب نماید.

برای روشن شدن بهتر موضوع مثالی را بررسی میکنیم. در این مثال کلاسی ایجاد میشود که در آن آرایه ای که به نوع خاصی محدود شده است، مورد استفاده قرار گرفته است.

```
public class TypeSafeList<T>
{
    T[] innerArray = new T[0];
    int currentSize = 0;
    int capacity = 0;
    public void Add(T item)
    {
        // see if array needs to be resized
        if (currentSize == capacity)
        {
            // resize array
            capacity = capacity == 0 ? 4 : capacity * 2; // double capacity
            T[] copy = new T[capacity]; // create newly sized array
            Array.Copy(innerArray, copy, currentSize); // copy over the array
            innerArray = copy; // assign innerArray to the new, larger array
        }
        innerArray[currentSize] = item;
        currentSize++;
    }
    public T this[int index]
    {
```

```
get
{
    if (index < 0 || index >= currentSize)
        throw new IndexOutOfRangeException();
    return innerArray[index];
}

set
{
    if (index < 0 || index >= currentSize)
        throw new IndexOutOfRangeException();
    innerArray[index] = value;
}

}

public override string ToString()
{
    string output = string.Empty;
    for (int i = 0; i < currentSize - 1; i++)
        output += innerArray[i] + ", ";
    return output + innerArray[currentSize - 1];
}

}
```

در نخستین خط این کد و درون کلاس **TypeSafeList**، شناسه ای جهت تعیین نوع **T** تعریف شده است. این کد معین میکند که استفاده کننده از آن میبایست یک نوع خاص را برای کار با آن در نظر بگیرد. در این کد، این نوع قابل انتخاب توسط برنامه

نویس، با عنوان **T** تعریف شده که میتوان به جای **T** از هر نام متغیر دیگری نیز استفاده کرد. از این متغیر در متدها و ویژگیهای کلاس نیز استفاده شده است. برای مثال، آرایه داخلی از نوع **T** تعریف شده و متد **Add** متغیری از نوع **T** را بعنوان ورودی میپذیرد.

برای تعریف متغیری از این کلاس، برنامه نویس میبایست نوع **T** را معین نماید :

```
TypeSafeList<type> variableName;
```

در قطعه کد زیر نیز، نمونه ای از کلاس **TypeSafeList** که نوع **int** را میپذیرد نشان داده شده است :

```
TypeSafeList<int> fib = new TypeSafeList<int>();
```

```
fib.Add(1);
```

```
fib.Add(1);
```

```
for (int i = 2; i < 25; i++)
```

```
fib.Add(fib[i - 2] + fib[i - 1]);
```

```
Console.WriteLine(fib.ToString());
```

از مهمترین مزایای **Generics** به شرح زیر است :

Type-Safe : با استفاده از کلاس **SafeTypeList**، برنامه نویس اطمینان حاصل میکند که از انواع ناخواسته به طور اشتباه استفاده نخواهد کرد.

کارآیی : کنترل نوع مورد استفاده در زمان اجرا را حذف کرده و هزینه مربوط به **Boxing** و **Unboxing** را کاهش میدهد.

قابلیت استفاده مجدد : قابلیت استفاده مجدد در برنامه های مختلف را بالا میبرد.

بسیاری از ساختمان داده‌هایی که در این سری مطالب مورد بررسی قرار خواهند گرفت از Generics ها استفاده میکنند و بهنگام ساخت یک ساختمان داده، مانند درخت دودویی که در قسمت سوم مورد بررسی قرار خواهد گرفت، از Generics استفاده خواهیم کرد.

لیست: آرایه ای همگون با قابلیت تغییر سائز ابعاد

یک آرایه، همانطور که قبلا نیز مشاهده کردیم، ساختمان داده ای با قابلیت نگهداری تعداد معینی عنصر هم نوع است که بطور مرتب در کنار یکدیگر قرار گرفته اند. هر چند استفاده از آرایه بسیار ساده است، اما در مواردی همچون زمانی که تعداد عناصر آرایه برای ما مشخص نیست و یا نیاز به تغییر سائز آرایه وجود داشته باشد، میتواند مشکل ساز باشد. یک روش برای پرهیز از تغییر سائز آرایه بطور دستی، ایجاد ساختمان داده ای مشابه به آرایه است که علاوه بر امکان خواندن و نوشتن اعضاء، قابلیت افزایش سایر را نیز در موارد مورد نیاز فراهم نماید. راهکار Net Framework. برای این ساختمان داده، لیست است که در Sysyem.Collections.Generic قرار دارد.

لیست، کلاسی است که درون خود از یک آرایه بهره می برد و علاوه بر قابلیت خواندن و نوشتن اعضاء این آرایه داخلی، قابلیت تغییر سائز خودکار آرایه را نیز فراهم می نماید. لیست نیز همانند آرایه، ساختمان داده ای همگون است، بدین معنا که تنها قابلیت ذخیره سازی عناصر هم نوع (از یک نوع داده ای) را دارا می باشد. لیست، از Generics بهره میبرد تا با استفاده از آنها، برنامه نویس امکان تعیین نوع داده مورد نظر را در هنگام استفاده، دارا باشد. در هنگام ایجاد نمونه ای جدید از کلاس لیست، میبایست نوع داده مورد استفاده در آن تعیین گردد:

// ساخت لیستی از نوع داده ای int

```
List<int> myFavoriteIntegers = new List<int>();
```

// ساخت لیستی از نوع داده ای رشته

```
List<string> friendsNames = new List<string>();
```

توجه کنید که نوع داده مورد استفاده در لیست بهنگام ایجاد نمونه ای جدید از کلاس لیست مشخص می گردد. بهنگام ایجاد نمونه جدید از لیست نیازی به تعیین سائز لیست نیست، هر چند میتوان سائز پیش فرضی برای لیست در نظر گرفت. برای افزودن عنصری به لیست، تنها کافیست تا از متد `Add()` آن استفاده کنید. همانند آرایه، دسترسی به عناصر لیست نیز از طریق اندیس صورت می گیرد. در مثال زیر لیستی از `int` ایجاد شده است. سپس با استفاده از متد `Add()` عناصری به آن افزوده شده و سپس با استفاده از اندیس به این عناصر دسترسی پیدا شده است.

// ساخت لیست

```
List<int> powersOf2 = new List<int>();
```

// افزودن ۶ عنصر به لیست

```
powersOf2.Add(1);
```

```
powersOf2.Add(2);
```

```
powersOf2.Add(4);
```

```
powersOf2.Add(8);
```

```
powersOf2.Add(16);
```

```
powersOf2.Add(32);
```

// تغییر دومین عنصر لیست به مقدار ۱۰

```
powersOf2[1] = 10;
```

// انجام یک عملیات ریاضی با استفاده از عناصر لیست

```
int sum = powersOf2[2] + powersOf2[3];
```

لیست، در حقیقت پیدا سازی خاص از یک آرایه است که پیچیدگی های پیاده سازی را پنهان کرده است. با استفاده از لیست، نیازی به تعیین سائز آرایه در ابتدا ایجاد آن نداریم. همچنین به هنگام افزودن عنصری به آن، نیازی به تغییر سائز آرایه داخلی آن نیست. علاوه بر مطالب گفته شده، لیست متدهایی جهت انجام عملیات مورد استفاده بر روی آرایه ها می باشد. برای مثال، در یک آرایه، برای پیدا کردن یک عنصر درون اعضای آرایه می بایست از حلقه `for` استفاده نمایید. در لیست با استفاده از متد

Contains() می توان به وجود عنصری خاص در لیست پی برد. همچنین با استفاده از متد **IndexOf()** می توان به محل ذخیره سازی عنصری خاص در لیست پی برد. کلاس لیست همچنین دارای متد **BinarySearch()** است که با استفاده از آن می توان به جستجوی باینری درون لیست مرتب شده پرداخت. با استفاده از متدهای دیگری نظیر **Find()** ، **FindAll()** ، **Sort()** و **ConvertAll()** نیز می توان با استفاده از **delegate** ها، به اجرای عملیاتی پرداخت که اجرای آنها بر روی آرایه ها نیاز به نوشتن چندین خط کد دارد.

زمان اجرای لیست، همانند زمان اجرایی یک آرایه استاندارد است، هر چند لیست توان پردازشی بیشتری نسبت به آرایه نیاز دارد.

نتیجه گیری:

در این مطلب، به عنوان اولین مطلب از سری مطالب مربوط به ساختمان های داده، با مفهوم ساختمان داده و اهمیت استفاده از آنها و همچنین با نحوه آنالیز کارایی یک ساختمان داده و اهمیت آن، آشنا شدیم.

پس از آشنایی با چگونگی آنالیز کارایی ساختمان های داده، به بررسی دو ساختمان داده مهم و پر کاربرد در **.Net** **Framework Base Class Library** یعنی **System.Array** و **System.Collections.Generic.List** پرداختیم. آرایه ساختمان داده ای همگون بود که با استفاده از آن به سادگی می توانیم به عناصر آن دسترسی پیدا کرده و عمل خواندن و نوشتن را انجام دهیم. نقطه ضعف آرایه در جستجوی عنصری خاص در آن بود. همچنین افزایش سایز آرایه نیز نیازمند نوشتن چندین خط کد بود.

کلاس لیست، با استفاده از چند متد، کارایی آرایه را افزایش داد. برای مثال متد **Add()** عنصری را به لیست اضافه می کند و سایز لیست را بطور خودکار افزایش می دهد. همچنین با استفاده از متد **IndexOf()** می توان اندیس عنصری خاص در لیست را بدست آورد. زمان اجرایی لیست چندان تفاوتی با آرایه ندارد اما کار برنامه نویس را در پیدا سازی بسیار آسان می نماید.

در قسمت بعدی با کلاس پشته (**Stack**) و صف (**Queue**) آشنا می شویم. در قسمت بعد همچنین با آرایه های شرکت پذیر آشنا می شویم. این آرایه ها از کلید هایی رشته ای بعنوان اندیس استفاده می کنند و در **.Net Framework** با استفاده از کلاسهای **Dictionary** و **Hashtable** ایجاد شده اند.

این مطالب خارج از فهرست بندی کتاب است...

مطالبی که در این قسمت خواهید دید :

شیء چیست؟

ارث‌بری یا Inheritance

انتزاع یا Abstraction

کیسوله کردن یا Encapsulation

چندریختی یا Polymorphism

منابع این مطلب

مفاهیم شیء گرایی در C#

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

شیء چیست؟

همانطور که گفتیم، با یک دید تصویری به سادگی می توانید اشیاء مختلفی را در اطراف خود بیابید. تمامی این اشیاء دارای سطوح و درجه پیچیدگی متفاوتی هستند. پیچیدگی آنها به شکل ظاهری و نوع رفتار آنها بستگی دارد.

در شیء گرایی به "شکل ظاهر" در اصطلاح، صفت یا Attribute و به عملی که شیء انجام می دهد، رفتار یا Behavior می گویند.

برای مثال یک صندلی را در نظر بگیرید. صندلی صفات مختلفی دارد ولی رفتار خاصی ندارد. مثلاً پایه های صندلی جزو صفات آن بشمار می روند. با کمی دقیق تر شدن می توان از این صفات برای توصیف صندلی استفاده کرد. بعنوان مثال تعداد پایه های صندلی می تواند عددی بین ۳ تا ۵ باشد. محل نشستن صندلی می تواند جمله‌ای در وصف جنس آن و مقدار مصرف ماده سازنده آن باشد. پستی صندلی را نیز می توان بعنوان متغییری boolean در نظر گرفت چراکه برخی از صندلی ها فاقد پستی هستند. با استفاده از این سه صفت ساده، به راحتی می توان صندلی را توصیف نمود و با همین سه صفت میتوان گونه‌های مختلفی از صندلی را نیز توصیف کرد.

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

حال شیء دیگری مانند تلویزیون را در نظر بگیرید. صفاتی که می توان برای تلویزیون در نظر گرفت عبارتند از : صفحه نمایش، سازنده آن و ... برای تلویزیون به راحتی می توان رفتار در نظر گرفت : خاموش و روشن شدن، تغییر کانال و کم و زیاد کردن صدا. این رفتارها بر اثر درخواست یک انسان یا همان کاربر اتفاق می افتند. شیء تلویزیون را میتوان بصورت زیر نمایش داد :

تلویزیون
سازنده
اندازه صفحه نمایش
بخش صدا()
تغییر کانال()
خاموش و روشن شدن()

بطور کلی، یک شیء را میتوان بوسیله صفات و رفتارهای آن بطور کامل توصیف نمود. یک شیء حتماً جسمی فیزیکی نیست، بلکه هر چیز قابل تصویری است که دارای صفت و رفتار است. در حقیقت میتوان گفت یک شیء شبیه به یک اسم است. اگر بتوان برای این اسم، صفت و رفتاری تعریف کرد، آن وقت تبدیل به شیء می شود.

از دید انتزاعی، زمان را نیز می توان بعنوان یک شیء در نظر گرفت. صفات زمان، ساعت، دقیقه و ثانیه هستند و گذشت زمان، رفتار آن است. در ایجاد شیء هیچ محدودیتی وجود ندارد و همه چیز به تخیل شما باز میگردد.

در زبان C#، اشیاء بوسیله کلاسها (Class) نمایش داده می شوند. داخل کلاس، صفات بصورت فیلدها ظاهر می شوند و جهت پیاده سازی رفتارها از متدها استفاده می گردد. به مثال زیر توجه نماید :

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

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

در این مثال، کلاس Time مشاهده می شود. این کلاس با کلمه کلیدی class اعلان گردیده است. همانطور که میدانید، دو گروه باز و بسته {} نیز ابتدا و انتهای کلاس را مشخص می کند. فیلدها دارای نام و نوع هستند. متدها دارای نام و نوع بازگشتی می باشند و پیاده سازی آنها داخل بلوک مربوط به خود آنها (بین دو {}) انجام میگردد.

یک شیء می تواند هر یک از نیازمندی های یک پروژه باشد. طراحی و اعلان صحیح اشیاء و مشخص کردن صفات و رفتار آنها یکی از مقوله های مهم در مهندسی نرم افزار بر پایه شیء گرایی است، چراکه همگی تراکنشها بین اشیاء صورت می پذیرند.

کلاس کردن اشیاء (مقدمه ای بر ارث بری Inheritance)

طبقه بندی اشیاء در گروه های مختلف بسیار سودمند است. زمین شناسان سنگها را طبقه بندی می کنند و زیست شناسان گیاهان و حیوانات را طبقه بندی می کنند. طبقه بندی اشیاء باعث می شود تا با دقت و ظرافت بیشتری بتوان به جزئیات هر طبقه و یا هر نوع پرداخت.

برای مثال حیوانات را در نظر بگیرید. زیست شناسان حیوانات را به دسته های پرندگان، مهره داران و خزندگان تقسیم می کنند. پرندگان صفاتی همچون منقار و بال دارند، مهره داران بدنی مودار دارند و خون گرمند و خزندگان خون سردند. از دید رفتاری نیز

پرنندگان پرواز می کنند، مهره داران به بچه های خود شیر می دهند و خزندگان می خزند. اینها ویژگیهای ارشد یا عام هستند و صفات دیگری در زیر گروهها افزوده می شوند. در گروه اصلی تنها صفات و رفتارهایی قرار می گیرند که عمومیت داشته باشند و سایر صفات و رفتارها در زیر گروهها معین می گردند. بعنوان مثال کلاس پرنندگان را بشکل زیر میتوان پیاده سازی نمود:

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

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

می توان این کلاس را بعنوان کلاسی عمومی برای پرنندگان در نظر گرفت که دارای فیلدی جهت تعیین نوع پرنده نیز می باشد. با استفاده از فیلد typeOfBird می توان گونه پرنده مورد نظر را معین نمود.

سلسله مراتب اشیاء (بررسی ارث بری در محیط شیءگرا)

روش گفته شده در بالا، جهت افزودن صفات برای یک طبقه بندی ساده کارآمد است، اما برای طبقه بندی های پیچیده قابل قبول نیست. تصور کنید حجم عظیمی نیازمندی در پروژه وجود داشته باشد و به سطح های مختلفی از طبقه بندی نیاز داشته باشید. با افزودن صفات جدید به هر یک از طبقات و سطحها به پیچیدگی پروژه به شدت افزوده می شود.

اتفاقی که در مورد زیر شاخه ها رخ می دهد، ایجاد یک سلسله مراتب طبیعی است. برنامه نویسی شیءگرا، متدولوژی را جهت مدیریت سلسله مراتب طبیعی فراهم می نماید. بعنوان مثال، اگر در یک پروژه یکی از طبقه بندیهای ما حیوانات باشند، منطقی است که شیءای از طبقه حیوانات در رأس سلسله مراتب قرار گیرد و در طبقات و زیر شاخه های سطوح پائینی، پرنندگان، مهره داران و خزندگان قرار گیرند. در سطح بعدی می توان چند پرنده مانند اردک، کلاغ و ... را نیز در زیر شاخه پرنندگان قرار داد. این سلسله مراتب تا سطح مورد نظر قابل تعمیم است.

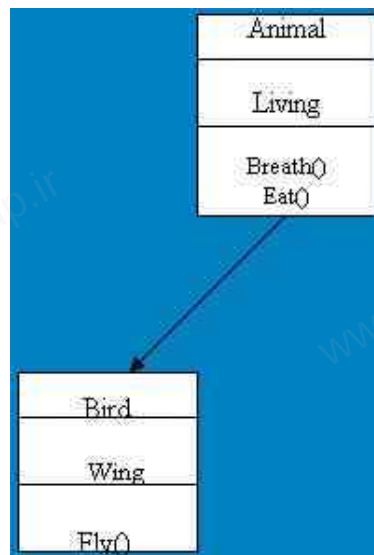
در برنامه نویسی شیءگرا، مفهومی که اشیاء را تحت یک سلسله مراتب خاص قرار میدهد، ارث بری یا Inheritance نامیده می شود. مثلاً طبقه بندی حیوانات را در نظر بگیرید، شیءای که در بالاترین سطح قرار می گیرد، شیء Animal است. این شیء دارای ویژگیهایی بسیار کلی و عمومی است و باید چنین هم باشد، چراکه سرگروه باید همیشه ویژگیهایی را داشته باشد که در زیر شاخه ها بطور مشترک وجود دارد و هر یک از زیر شاخه ها به ویژگیهای سرگروه خود، ویژگیها و مشخصات خاص خود را می افزایند.

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

تفاوت Animal با سایر اشیاء که در این سلسله مراتب قرار می‌گیرند در آنست که سایر اشیاء می‌بایست صفاتی دیگر و - یا رفتارهای دیگری را نیز به صفات و رفتارهای Animal اضافه کنند. این صفات و رفتارها مسلماً جزئی‌تر و دقیق‌تر از صفات و رفتارهای سرگروه است. همین مسئله مفهومی را در پیمایش سلسله مراتب‌ها بوجود می‌آورد: در پیمایش بالا به پائین (Top-Down) سلسله مراتب‌ها به جزئیات یا گونه‌های خاص برخورد می‌کنیم، حال آنکه در پیمایش پائین به بالا (Bottom-Up) به گروه‌ها و دسته‌های عمومی می‌رسیم.

اشیاء سطوح پائینی (که به آنها child می‌گوئیم) صفات و رفتارهای اشیاء سطح بالاتر خود را به ارث می‌برند. به این اشیاء بالاتر یا سرگروه‌ها نیز parent می‌گوئیم. به این رابطه موجود بین child و parent در اصطلاح رابطه "هست" یا "بودن" (is-a relationship) می‌گویند. مثلاً می‌گوئیم "اردک یک پرند است".

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



همانطور که در این شکل ملاحظه می‌کنید، سرگروه که همان Animal است، دارای صفت Living و رفتارهایی همچون Eat() می‌باشد. زیر شاخه این شیء کلیه صفات آنرا به ارث می‌برد، بعلاوه اینکه صفت Wing و رفتار Fly() را نیز به صفات و رفتارهای parent خود می‌افزاید. حال برای زیر شاخه‌ای که در زیر Bird قرار می‌گیرد نیز می‌توان صفات و رفتارهای خاص دیگری در نظر گرفت.

علاوه بر طبقه‌بندی اشیاء در دسته‌های مختلف و سادگی در سازماندهی آنها بوسیله ارث‌بری، استفاده از ارث‌بری در انجام کارها نیز صرفه‌جویی ایجاد می‌کند. هر شیء جدیدی که به یک سلسله مراتب وارد می‌شود، بطور خودکار تمامی صفات و رفتارهای کلیه parent های خود را دارا می‌باشد و بعثت ارث‌بری نیازی به تعریف مجدد این صفات برای شیء جدید نمی‌باشد. به بیان دیگر می‌توان گفت، ارث‌بری روشی برای استفاده مجدد از صفات و رفتارهای موجود است.

استفاده از ارث‌بری این امکان را برای طراحان نرم افزار فراهم می‌کند تا وقت بیشتری برای تفکر بر روی منطق برنامه صرف کنند و درگیر پیچیدگی‌های پیاده‌سازی و نگهداری نرم افزار نشوند.

انتزاع (Abstraction)

اکنون زمان مناسبی برای بحث درباره انتزاع است. برخی اشیاء تا حدودی انتزاعی هستند و برخی دیگر کاملاً واقعی. بعنوان مثال، چیزی بعنوان Animal وجود ندارد، بلکه این تنها توصیف کلاسی از اشیاء است. همچنین موجودیتی وجود ندارد که از لحاظ فیزیکی یک پرنده باشد. این تنها طبقه‌بندی و دسته‌بندی است که مورد استفاده قرار می‌گیرد.

از طرف دیگر شی‌ای وجود دارد بنام اردک که واقعاً یک اردک است و دارای کلیه صفات و رفتارهای آن می‌باشد. البته باید توجه داشت که هر اردک تنها نمونه‌ای از "اردک" است. (منظور در اینجا آنست که موجودیتی مانند یک اردک که واقعا وجود دارد، خود نمونه‌ای (instance) از کلاس اردک است. درک مطالب در اینجا شاید به اندکی تأمل و حوصله نیاز داشته باشد!)

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

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

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

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

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

اشیاء درون اشیاء (مقدمه‌ای بر کپسوله کردن یا Encapsulation)

ساختارهای سلسله مراتبی روشی جهت دستیابی به روابط بین اشیاء هستند. هر چند روشهای دیگری نیز برای نشان دادن روابط بین اشیاء وجود دارد. یکی از این روشها که بسیار معمول نیز هست، استفاده از اشیاء درون اشیائی دیگر است.

یکی از اصطلاحات رایج برای بیان این مفهوم، کپسوله کردن یا Encapsulation است. استفاده صحیح از این مفهوم باعث کاهش پیچیدگی شده و تنها آن قسمت اطلاعات و جزئیات را نمایان می‌کند که برای درک یک شی‌ای لازم است. بعنوان مثال، پرواز

پرنندگان مسئله‌ای است که توجه بسیاری از زیست‌شناسان را به خود جلب کرده است. آنها برای درک پرواز، ساختار بالها، پرها و اسکلت پرنندگان را مورد مطالعه قرار می‌دهند. برای درک پرواز، بررسی همین اجزاء و جزئیات کافیت و نیازی به مطالعه درباره سیستم گوارش پرنندگان نمی‌باشد. برای بررسی یک موضوع تنها اطلاعاتی مربوط به آن در حل و بررسی مسئله می‌تواند موثر باشد و اطلاعات اضافی، نه تنها وقت‌گیر و بی‌فایده است، بلکه می‌تواند گیج‌کننده نیز باشد و بر پیچیدگی مسئله، بدون دلیل، بیفزاید. با استفاده از کپسوله‌کردن تنها آن قسمت از اطلاعات که مورد نظر است در دسترس قرار می‌گیرند.

شیء Bird را که قبلاً درباره آن صحبت کردیم، در نظر بگیرید. پرنندگان دارای منقار و بال هستند. خود بال یک شیء است که دارای صفاتی نظیر پر و اندازه است. رفتار آن نیز می‌توان باز و بسته شدن در حین پرواز باشد. نکته قابل اشاره در اینجا، کلمه "داشتن" است. اعمالی وجود دارند که پرنده آنها را انجام می‌دهد و این اعمال خود جزئی از پرنده هستند.

بطور کلی، رابطه "داشتن" (has-a relationship) بین یک شیء و اعمالی که بر روی اجزای خود انجام می‌دهد، وجود دارد. بعنوان مثال می‌گوئیم "پرنده بال دارد". در اینجا چون بال شیء‌ای متعلق به پرنده است که پرنده روی آن عملی انجام می‌دهد، مفهوم کپسوله‌کردن رخ می‌دهد، شیء‌ای درون شیء‌ای دیگر. مثال زیر نحوه پیاده‌سازی کپسوله‌کردن را نشان می‌دهد.

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

    void flap()
    {
        //implementation
    }

    void fold()
    {
        //implementation
    }
}

class Bird : Animal
{
    int beakSize;
    Wing wings;

    void Fly()
    {
        //implementation
    }
}
```

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

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

در برخی شرایط ممکن است نیاز داشته باشیم تا علاوه بر اینکه شیء ای را در رده خاصی طبقه بندی می کنیم، به آن شیء این اجازه را نیز بدهیم تا بتواند رفتارهای خاص به خود را داشته باشد.

برای مثال، اردک و یا عقاب اشیایی هستند که با گروه Bird ارتباط دارند. تصور کنید زیست شناسی می خواهد رفتارهای این پرندگان را بررسی نماید. برای بررسی این پرندگان لازم است تا زیست شناس آنها را در جایی نگهداری کرده و بر روی آنها مطالعه انجام دهد. مسلماً این پرندگان باید در قفس نگهداری شوند اما مطمئناً قفس اردک با قفس عقاب یکسان نمی تواند باشد. این بدین معناست، قفسی را که تعریف می کنیم باید طوری باشد تا برای اکثر گونه های پرندگان مناسب باشد. می توانیم قفسی مخصوص پرندگانی که پرواز می کنند در نظر بگیریم. از اینرو پرندگانی را که پرواز می کنند را در این قفس قرار می دهیم. همانطور که مشاهده می کنید، دسته بندی پرندگان در یک گروه جهت رسیدن به اهداف پروژه ضروری است. پیاده سازی مطلب گفته شده بشکل زیر است.

```
abstract class flyingBird : Bird
{
    //implementation
}

class Eagle : flyingBird
{
    //implementation
}

class Duck : flyingBird
{
    //implementation
}

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

در کد فوق، ۳ کلاس جدید ایجاد شده است. اولین کلاس، flyingBird است که از کلاس Bird مشتق شده است. از آنجائیکه تمامی پرندگان پرواز نمی کنند، این کلاس می تواند تنها مخصوص آن دسته از پرندگانی باشد که می توانند پرواز نمایند. کلاسهای Eagle و Duck در این کد از FlyingBird مشتق شده اند. آخرین کلاس نیز Experiment است که متد Main() در آن قرار گرفته است. درون متد Main()، آرایه ای بنام flyingBirdCage از نوع FlyingBird تعریف شده که این آرایه، مخصوص اشیایی از نوع FlyingBird است. از آنجائیکه کلاسهای Duck و Eagle از نوع FlyingBird هستند، پس می توان آنها را در این آرایه قرار داد. نکته مهم در اینجا نیز همین مطلب است که از این پس ما پرندگانی داریم که پرواز می کنند و در قفس

مخصوص نگهداری می‌شوند. حال تصور کنید که این نوع قفس وجود نداشت و مجبور بودیم تا برای هر پرنده نوع خاصی از قفس را تعریف نمائیم. در این شرایط حجم کار بسیار زیاد می‌شد و به مشکل بر می‌خوردیم. اما هم اکنون می‌دانیم که قفسی داریم که شرایط کلی و عمومی برای نگهداری پرندگانی که پرواز می‌کنند را داراست. در صورتیکه برای هر پرنده بخواهید قفسی مخصوص ایجاد کنید، کدی شبیه به کد زیر مورد نیاز بود :

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

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

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

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

```
using System;

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

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

class Duck : FlyingBird
```

```

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

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

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

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

```

خروجی این برنامه بشکل زیر است :

```

Eagle Fly
Duck Fly

```

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

به همین ویژگی، یعنی داشتن یک رفتار در گروهی از اشیاء که هر یک از آنها این رفتار را بصورت دلخواه خود پیاده‌سازی می‌کنند، Polymorphism گفته می‌شود. استفاده از Polymorphism در حل مسایل پیچیده بسیار سودمند است.

پایگاه داده در سی شارپ - بخش اول

پیش گفتار

در این مطلب، که آغاز سری مطالب مرتبط با پایگاه های داده و نحوه برقراری ارتباط با آنها است، در ابتدا به معرفی مختصری از پایگاه داده پرداخته می شود و سپس مقدمات کار با پایگاه داده آموزش داده شده است. از آنجائیکه برقراری ارتباط با پایگاه داده نیازمند داشتن دانشی از زبان SQL است، از اینرو در این سری مطالب، ابتدا به بیان دستورات بنیادی SQL پرداخته شده و سپس نحوه استفاده از آنها بیان شده است. برای سادگی کار و جلوگیری از پیچیده شدن مطالب در ابتدا کار، در مطالب نخستین از پایگاه داده MS Access استفاده شده است که در بسته نرم افزاری MS Office قرار دارد. پس از بررسی مقدمات و آموختن نحوه برقراری ارتباط با یک پایگاه داده، به سراغ مطالب پیشرفته خواهیم رفت و با بسط مطالب و عمیق شدن در زبان SQL، نحوه کار با MS SQL Server 2000 را نیز فرا خواهیم گرفت. در دروس ابتدایی بیشتر سعی شده تا برای یادگیری بهتر و سادگی مطالب، قسمتی از کد نویسی های مربوط به برقراری ارتباط با پایگاه داده را کم کرده و این کار را به Visual Studio IDE محول سازیم اما در ادامه کار، با تمامی قسمتها و کد نویسی کامل آشنا خواهیم شد.

برای شروع کار، به MS Access (ترجیحا نسخه ۲۰۰۳) و Visual Studio .Net (بهتر است از نسخه ۲۰۰۳ استفاده کنید اما تمامی کدها با نسخه ۲۰۰۲ نیز قابل اجرا هستند اما توجه نمایید که در صورت استفاده از Visual Studio .Net 2005 و یا Visual C# Express Edition برای اجرای برخی کدها دچار مشکل خواهید شد.)

مقدمه

پایگاه داده یا همان Database، مجموعه ای سازمان داده شده از اطلاعات است که به شکل جدا/ول ذخیره می شوند. برای سازمان دهی این اطلاعات روشهای متفاوتی وجود دارد که هدف تمامی آنها فراهم کردن روشهایی مناسب برای سهولت در برقراری ارتباط با پایگاههای داده و استفاده از اطلاعات موجود در آنها است. سیستم مدیریت پایگاه داده (DBMS)، مکانیزمی را جهت ذخیره سازی و بازیابی اطلاعات در پایگاه داده فراهم می نماید. در حقیقت DBMS باعث می شود تا برنامه نویس بدون نگرانی درباره چگونگی ذخیره سازی داده ها در پایگاه داده و ساختار آنها، به اطلاعات دسترسی پیدا کند و بتواند داده های جدید را در آن ذخیره نماید.

امروزه اکثر پایگاه داده های محبوب از نوع *رابطه ای (Relational)* هستند. همانند پایگاه داده های عادی، در پایگاه داده های رابطه ای نیز دسترسی به اطلاعات ذخیره شده در جداول از طریق زبان پرس و جوی ساخت یافته یا همان SQL میسر میگردد که زبانی استاندارد است و توسط اکثر نرم افزارهای مرتبط با پایگاه داده مورد استفاده قرار می گیرد. از جمله سیستمهای پایگاه داده رابطه ای می توان به MS SQL Server، Oracle™، Sybase™، DB2™، Informix™ و MySQL™ اشاره کرد.

زبانهای برنامه نویسی از طریق یک *Interface* (یا همان نرم افزاری که ارتباط بین DBMS و یک برنامه را فراهم می کند) به پایگاه داده متصل شده و با آنها به تعامل میپردازند. در C# برقراری ارتباط با پایگاه داده از طریق ADO.Net انجام میشود. ADO.Net در حقیقت رابط بین نرم افزار و پایگاه داده است و امکانات ویژه ای را جهت دسترسی به اطلاعات موجود در آن در اختیار برنامه نویس قرار می دهد.

مدل پایگاه داده رابطه ای (Relational Database Model)

مدل پایگاه داده رابطه ای، نمایش منطقی ای از داده هاست که رابطه موجود بین داده ها را، بدون درگیر شدن با ساختار فیزیکی آنها، نشان میدهد. پایگاه داده رابطه ای از جداول تشکیل میشود. هر جدول خود از سطرها یا رکوردها (record/row) و ستونها یا فیلدها (columns/fields) تشکیل میشود. در شکل ۱، جدول نمونه ای به نمایش گذاشته شده است که نام آن Employee است و هدف آن نمایش دادن اطلاعات پرسنلی کارمندان یک اداره است. این جدول از ۶ رکورد تشکیل شده و فیلد number متعلق به هر رکورد در آن *Primary Key* است که جهت ارجاع به داده ها در جدول مورد استفاده قرار می گیرد. *Primary Key* فیلد یا فیلدهایی در جدول است (هستند) که شامل داده هایی یکتا باشند، بدین معنا که هیچ رکورد دیگری مقداری مشابه با آن ندارد. با استفاده از مفهوم *Primary Key*، این تضمین وجود دارد که هر رکورد حداقل با یک مقدار یکتا قابل شناسایی است. یک مثال خوب برای فیلد *Primary Key*، فیلدی است که حاوی کد ملی افراد باشد، چراکه در یک جامعه به ازای هر شخص یک کد ملی یا یک شماره منحصر بفرد وجود دارد و هیچ دو شماره ملی مشابهی یافت نمی شود. در جدول نمونه ما، فیلد *Primary Key* حاوی شماره پرسنلی کارمندان است. همانطور که مشاهده می شود اطلاعات جدول بر اساس فیلد number مرتب شده است. در اینجا، اطلاعات بصورت صعودی مرتب شده اند. (حالت دیگر مرتب شدن حالت نزولی است).

هر ستون از جدول، فیلدی متفاوت را نشان میدهد. معمولاً رکوردها در جدول منحصر بفرد هستند (بوسیله *Primary Key*) اما مقادیر فیلدهای مختلف می تواند مشابه با یکدیگر باشد. برای مثال، ۳ رکورد مختلف در فیلد Department از جدول Employee حاوی مقدار ۴۱۳ هستند.

number	name	department	salary	location
۲۳۶۰۳	میثم	۴۱۳	۱۰۰۰	تهران
۲۴۵۶۸	علی	۴۱۳	۱۵۰۰	تهران
۳۴۵۸۹	محمد	۶۴۲	۱۳۰۰	مشهد
۳۵۷۶۱	ناصر	۶۱۱	۱۸۰۰	اصفهان
۴۷۱۳۲	مریم	۴۱۳	۱۵۰۰	تهران
۷۸۳۲۱	فاطمه	۶۱۱	۲۵۰۰	اهواز

با توجه به اینکه حجم اطلاعات قابل ذخیره در یک جدول نامحدود است، از اینرو باید بتوان با استفاده از روشی تنها به آن قسمت از اطلاعات دسترسی پیدا کرد که مورد نظر کاربر است. برای این منظور از SQL استفاده می نماییم. SQL مجموعه دستوراتی را فراهم می نماید که با استفاده از آنها قادر خواهیم بود تا اطلاعات مورد نظر را از پایگاه داده انتخاب (SELECT) کرده و مورد استفاده قرار دهیم.

نگاهی بر پایگاه داده رابطه ای

در ادامه مطالب این قسمت، با استفاده از یک پایگاه داده نمونه بنام Book با دستورات اولیه SQL آشنا خواهیم شد. پایگاه داده مورد نظر ما از چهار جدول تشکیل شده است. این جداول به ترتیب حاوی اطلاعاتی درباره نویسنده کتاب (Authors)، ناشر کتاب (Publishers)، کد شناسایی نویسنده (AuthorISBN) و عنوان کتاب (Titles) هستند. جدول Authors از سه فیلد تشکیل شده است که عبارتند از شماره اختصاصی هر نویسنده، نام و نام خانوادگی. در جدول زیر مشخصات جدول Authors نشان داده شده است.

نام فیلد	توضیحات
authorID	شماره شناسایی نویسنده را در پایگاه داده مشخص مینماید. در پایگاه داده Book این فیلد از نوع int تعریف شده و بصورت فیلدی اعلان شده که بطور خودکار مقدارش یک واحد یک واحد اضافه میشود (Auto-Increment Field). با اضافه شدن هر رکورد به پایگاه داده مقدار

این فیلد یک واحد افزوده میشود که همین امر تضمین میکند که مقدار این فیلد همواره منحصر بفرد خواهد بود.	
نام نویسنده کتاب. (از نوع string)	firstName
نام خانوادگی نویسنده. (از نوع string)	lastName

Authors		
lastName	firstName	authorID
Deitel	Harvey	1
Deitel	Paul	2
Nieto	Tem	3
Steinbuhler	Kate	4
Santry	Sean	5
Lin	Ted	6
Sadhu	Praveen	7
McPhie	David	8
Yaeger	Cheryl	9
Zlatkina	Marina	10
Wiedermann	Ben	11
Liperi	Jonathan	12
Listfield	Jeffrey	13

جدول Publisher از دو فیلد تشکیل گردیده است که نمایش دهنده شماره شناسایی ناشر و همچنین نام ناشر است. شکل زیر مشخصات جدول Publisher را نشان میدهد.

نام فیلد	توضیحات
publisherID	شماره شناسایی ناشر را در پایگاه داده نشان میدهد. این فیلد که از نوع int است و بصورت خودکار مقدارش افزوده میشود، فیلد Primary Key جدول Publisher نیز میباشد.
publisherName	نام ناشر کتاب. (از نوع string)

Publishers	
publisherName	publisherID
Prentice Hall	1
Prentice Hall PTG	2

جدول AuthorISBN شامل دو فیلد است که نشان دهنده شماره شناسایی نویسنده و شماره ISBN کتابهایی است که یک نویسنده خاص آنها را نوشته است. با استفاده از این جدول میتوان کتابهایی را که یک نویسنده خاص آنها را نوشته بدست آورد. شکل زیر مشخصات جدول AuthorISBN را نشان میدهد.

نام فیلد	توضیحات
authorID	شماره شناسایی نویسنده کتاب را نشان میدهد. با استفاده از این فیلد میتوان کتابهای مرتبط با هر نویسنده را در پایگاه داده پیدا کرد. این فیلد که از نوع int میباشد، در جدول

Author نیز قرار دارد.	
شماره ISBN مربوط به هر کتاب. (string)	isbn

AuthorISBN	
authorID	ISBN
1	0130125075
2	0130125075
1	0130132497
2	0130132497
1	0130161438
2	0130161438
3	0130161438
1	0130284173
2	0130284173
3	0130284173
6	0130284173
7	0130284173
1	0130284181
2	0130284181
3	0130284181
8	0130284181

AuthorISBN	
authorID	ISBN
1	013028419X
2	013028419X
3	013028419X
1	0130293636
2	0130293636

جدول Titles از شش فیلد تشکیل شده است که عبارتند از ISBN مربوط به هر کتاب، عنوان کتاب، ویرایش، سال انتشار و شماره شناسایی ناشر، نام عکس جلد هر کتاب و قیمت هر کتاب. در شکل زیر مشخصات این جدول نشان داده شده است.

نام فیلد	توضیحات
isbn	شماره ISBN مربوط به هر کتاب. (string)
title	عنوان کتاب. (string)
editionNumber	شماره ویرایش کتاب (string)
copyright	سال نشر کتاب (int)
publisherID	شماره شناسایی ناشر (از نوع int). مقادیر این فیلد میبایست با شماره ای در جدول Publisher همخوانی داشته باشد.
imageFile	نام فایلی که عکس روی جلد کتاب در آن قرار دارد. (string)
price	قیمت کتاب. (از نوع اعداد حقیقی)

مشاهده جدول Titles

در شکل زیر رابطه بین جداول موجود در پایگاه داده BOOK نشان داده شده است. خط اول در هر جدول بیانگر نام جدول است. فیلدهایی که بصورت **Bold** نشان داده شده اند، بیانگر فیلدهای Primary Key هستند. همانطور که قبلا نیز گفته شد، فیلدهای Primary Key نمایش دهنده رکوردهایی منحصر بفرد در جدول هستند، از اینرو هر رکورد در جدول، میبایست دارای مقداری در فیلد Primary Key خود باشد و این مقدار میبایست منحصر بفرد باشد. از این اصل، بعنوان قانون "**جامعیت موجودیتها**" یاد میشود (Rule Of Entity Integrity). توجه نمایید که در جدول AuthorISBN دو فیلد بعنوان Primary Key نشان داده است، که این بدین معناست که در این جدول Primary Key مرکب وجود دارد، از اینرو هر رکورد در این جدول میبایست دارای authorID و isbn منحصر بفردی (بطور مشترک) باشد. برای مثال، ممکن است رکوردهای بسیاری در جدول یافت شوند که مقدار فیلد authorID آنها برابر با ۲ باشد، اما چون ترکیب authorID و isbn میبایست منحصر بفرد باشد، از اینرو هیچ دو رکوردی نباید یافت شود که دارای authorID و isbn مشابه باشد. همچنین ممکن است رکوردهای مختلفی مقدار فیلد isbn آنها برابر با ۰۱۳۰۸۹۵۶۰۱ باشد، اما تنها یک فیلد یافت میشود که مقدار authorID آن برابر با ۲ و مقدار isbn آن برابر با ۰۱۳۰۸۹۵۶۰۱ باشد.

توجه : در صورتیکه مقداری برای فیلد Primary Key انتخاب نشود، اصل جامعیت موجودیتها نقض شده، از اینرو DBMS خطایی را گزارش میکند.

توجه : وارد کردن مقادیر مشابه برای فیلد Primary Key باعث می شود تا DBMS خطایی را گزارش نماید.

خطهای رسم شده بین جداول، رابطه (Relationship) بین جداول را نشان می دهد. برای مثال خط رسم شده بین جداول Publisher و Titles را در نظر بگیرید. همانطور که مشاهده میشود، در سمت Publisher، بر روی خط عدد ۱ قرار دارد و در سمت Titles علامت ∞ (بینهایت) قرار گرفته است. خط رسم شده بین این دو جدول، بیانگر یک رابطه "**یک به چند**"

(One-to-Many) است که بیان میدارد، به ازای هر ناشر در جدول Publishers، تعداد زیادی کتاب می تواند در جدول Titles قرار داشته باشد. همچنین توجه داشته باشید که خط رسم شده بین این دو جدول، از فیلد publisherID در جدول Publisher آغاز شده و به فیلد publisherID در جدول Titles ختم شده است. فیلد publisherID در جدول Titles یک Foreign Key است، بدین معنا که به ازای هر موجودیت در این جدول، یک مقدار منحصر بفرد در جدولی دیگر وجود دارد و این مقدار منحصر بفرد فیلد Primary Key جدول دوم است. Foreign Key در زمان ایجاد یک جدول مشخص می شود. با استفاده از Foreign Key، اصل "جامعیت/ارجاع" (Rule Of Referential Integrity) مطرح می شود که بیان میدارد، مقدار هر Foreign Key باید در فیلد Primary Key جدولی دیگر وجود داشته باشد. با استفاده از Foreign Key میتوان اطلاعات موجود در جداول مختلف را با یکدیگر "پیوند" (join) زد و از آنها استفاده نمود. همواره رابطه ای "یک به چند" بین Primary Key و Foreign Key وجود دارد، بدین معنا که مقدار فیلد Foreign Key می تواند در جدول خودش چندین بار ظاهر شود اما در جدول دیگر تنها می تواند یکبار و آنهم بعنوان Primary Key ظاهر شود.

توجه : همانطور که گفته شد، همیشه رابطه ای "یک به چند" از سوی Primary Key به سمت Foreign Key وجود دارد.

توجه : استفاده از مقداری بعنوان فیلد Foreign Key که در Primary Key هیچ جدولی وجود نداشته باشد، اصل جامعیت ارجاع را نقض کرده، از اینرو DBMS خطایی را گزارش خواهد کرد.

زبان پرس و جوی ساخت یافته (SQL)

در این قسمت، به بررسی مختصر زبان SQL خواهیم پرداخت. همانند زبانهای برنامه سازی، زبان SQL نیز از کلمات کلیدی و دستوراتی تشکیل شده است که دستورات کلیدی و مهم آنرا در جدول زیر مشاهده می کنید.

نام دستور SQL	توضیحات
SELECT	فیلدهای مورد نظر را از جدول یا جداولی انتخاب می کند.
FROM	جداولی را مشخص می کند که اطلاعات از آنها باید انتخاب شود و یا قرار است اطلاعات

از آنها حذف گردد. در هر دستور SELECT و DELETE وجود دارد.	
شرایطی را مشخص می کند که تحت آن رکوردهایی خاص انتخاب خواهند شد.	WHERE
رکوردهای متفاوتی را از جداول مختلف به یکدیگر متصل مینماید و مجموعه ای جدید از رکوردها را ایجاد مینماید.	INNER JOIN
شرایطی را نشان میدهد که بوسیله آن دسته بندی اطلاعات انجام میشود.	GROUP BY
شرایطی را نشان میدهد که طی آن اطلاعات مرتب می شوند.	ORDER BY
داده را در جدولی خاص وارد می کند.	INSERT
داده خاصی را در جدول مورد نظر بروز رسانی می کند. (تغییر میدهد)	UPDATE
اطلاعات خاصی را از جدول مورد نظر حذف میکند.	DELETE

دستور SELECT

یکی از ساده ترین دستورات در SQL، دستوری است که در آن اطلاعات از جدولی خاص انتخاب می شود. چنین عملی با استفاده از دستور SELECT صورت میگیرد و ساده ترین فرمت این دستور بصورت زیر است :

```
SELECT * FROM tableName
```

که در آن علامت "*" بیان میدارد که تمامی اطلاعات (تمامی ستونها) موجود در جدول انتخاب شده و به نمایش در خواهند آمد و *tableName* نام جدولی است که میخواهیم اطلاعات را از آن انتخاب نماییم.

```
SELECT * FROM Authors
```

برای انتخاب فیلد یا فیلدهایی خاص از یک جدول، میتوانیم نام فیلدهای مورد نظر آن جدول را در جلوی دستور SELECT و بجای * قرار دهیم و آنها را با کاما "," از یکدیگر جدا نماییم.

```
SELECT authorID, lastName FROM Authors
```

تنها فیلد authorID و lastName از جدول Authors را نشان میدهد.

نکته : در صورتیکه نام فیلد شامل فضای خالی (Space) باشد، در اینصورت میبایست نام فیلد را داخل دو براکت "[]" قرار دهیم. (SELECT [author ID] FROM Authors)

دستور WHERE

در اکثر موارد کاربر در جدول به دنبال اطلاعاتی میگردد که دارای شرایط خاصی است. در این موارد تنها آن رکوردهایی که با شرایط مورد نظر کاربر همخوانی دارند میبایست نمایش داده شوند. SQL با استفاده از دستور WHERE در دستور SELECT، شرایط مورد نظر را اعمال می نماید. ساده ترین فرم دستور SELECT که به همراه WHERE باشد، بصورت زیر است :

```
SELECT fieldName1, fieldName2,... FROM tableName
WHERE criteria
```

که در آن *fieldName* نام فیلدهای مورد نظر، *tableName* نام جدولی که اطلاعات از آن استخراج می شود و *criteria* شرایطی است که بر اساس آن جستجو در جدول صورت میپذیرد. برای مثال، در صورتیکه بخواهیم عناوین کتابهایی را بیابیم که تاریخ نشر آنها بعد از سال ۱۹۹۹ است، از دستور SELECT زیر استفاده می کنیم :

```
SELECT title
FROM Titles
WHERE copyright > 1999
```

نکته : در زبان C# برای اجرای دستورات SQL از یک رشته استفاده می کنیم که این رشته حاوی کل دستور (Query) مورد نظر ما است. همچنین برای بالا رفتن خوانایی برنامه، دستورات SQL را در خطوط جدا مینویسیم.

دستور WHERE می تواند حاوی عملگرهای <, <=, >, >=, =, <> و عملگر LIKE باشد. عملگر LIKE برای "تطبیق الگو" (Pattern Matching) مورد استفاده قرار میگیرد و به همراه * و ? مورد استفاده قرار میگیرد. با استفاده از تطبیق الگو، میتوان به دنبال رشته ای گشت که حاوی الگوی مورد نظر است. برای مثال، query زیر، تمامی رکوردهایی از جدول Authors را نشان میدهد که lastName آنها با کاراکتر "D" شروع شده باشند :

```
SELECT * FROM Authors
WHERE lastName LIKE 'D*'
```

توجه نمایید، استفاده از کاراکتر * بیان میدارد که برای تطبیق الگو تنها کفایت تا کاراکتر اول مقدار D داشته باشد و سایر کاراکترها هر مقداری می توانند داشته باشند. همچنین اهمیتی ندارد که چه تعداد کاراکتر بعد از کاراکتر D در رشته وجود دارد و تنها اهمیت برای ما وجود کاراکتر D در ابتدای رشته است.

نکته : تمامی سیستم های پایگاه داده از عبارت LIKE پشتیبانی نمیکند.

نکته : در اکثر سیستم های پایگاه داده، به جای استفاده از کاراکتر "*" در عبارت LIKE، از کاراکتر "%" استفاده میشود.

نکته : در برخی از سیستم های پایگاه داده، کاراکترهای رشته، Case Sensitive هستند.

نکته : بهتر است برای تمیز دادن دستورات و عبارات SQL، آنها را با حروف بزرگ بنویسیم.

توجه نمایید در رشته ای که مورد تطبیق الگو قرار میگیرد، در صورتیکه کاراکتر "?" قرار گیرد، بدین معناست که بجای کاراکتر "?" تنها یک کاراکتر قرار خواهد گرفت. برای مثال در query زیر، تمامی رکوردهایی از جدول Authors نمایش داده می شوند که lastName آنها با هر کاراکتری شروع شده باشد و به دنبال آن کاراکتر "i" آمده باشد و بعد از آن هر تعداد کاراکتر قرار داشته باشد :

```
SELECT * FROM Authors
WHERE lastName LIKE '?i*'
```

نکته : اکثر پایگاه های داده، به جای استفاده از کاراکتر "?" از کاراکتر "_" در عبارت LIKE استفاده میکنند.

نکته : توجه نمایید که پس از عبارت LIKE، رشته مورد نظر درون یک جفت ' ' قرار میگیرد.

دستور ORDER BY

نتیجه یک query می تواند بصورت صعودی یا نزولی مرتب گردد. این کار با استفاده از دستور ORDER BY انجام میشود. ساده ترین فرم این دستور بصورت زیر است :

```
SELECT fieldName1, fieldName2, ... FROM tableName
ORDER BY field ASC
```

```
SELECT fieldName1, fieldName2, ... FROM tableName
ORDER BY field DESC
```

که در *field* نام فیلدی است که نتایج query بر اساس آن مرتب می شود. در صورتیکه بخواهیم نتایج بصورت صعودی مرتب شوند از کلمه ASC و در صورتیکه بخواهیم نتایج بصورت نزولی مرتب شوند از کلمه DESC استفاده می نماییم. بطور پیش فرض، نتایج query بصورت صعودی نمایش داده میشوند.

```
SELECT * FROM Authors
ORDER BY lastName DESC
```

همچنین، با استفاده از ORDER BY میتوان نتایج query را بر اساس چند فیلد مختلف نیز مرتب نمود :

```
ORDER BY field1 sortingOrder, field2 sortingOrder, ...
```

که در آن *field*، فیلدهایی هستند که مرتب سازی بر اساس آنها صورت میگیرد و *sortingOrder* نوع مرتب سازی هر فیلد را نشان میدهد. توجه نمایید، در مواردی که دو یا چند فیلد برای مرتب سازی در نظر گرفته شده اند، ابتدا مرتب سازی بر اساس فیلد اول انجام می شود و پس از آن مرتب سازی بر اساس فیلدهای دیگر به پیش میرود. در مثال زیر، ابتدا نتایج *query* بر اساس نام خانوادگی (*lastName*) و سپس بر اساس نام (*firstName*) مرتب میشوند :

```
SELECT * FROM Authors
ORDER BY lastName, firstName
```

توجه نمایید که در یک *query* می توان از ترکیب دستورات *WHERE* و *ORDER BY* استفاده نمود :

```
SELECT isbn, title, price FROM Titles
WHERE title LIKE '%How To Program'
ORDER BY title DESC
```

ادغام داده ها از جداول مختلف : INNER JOIN

طراحان پایگاه داده، معمولاً اطلاعات را به جداول مختلفی تقسیم میکنند تا مطمئن باشند که در پایگاه داده اطلاعات بیهوده و زائد ذخیره نمیشود. برای مثال، پایگاه داده *Book* شامل دو جدول *Authors* و *Titles* است. ما با استفاده از جدول *AuthorISBN*، لینکی بین نویسنده های مختلف و عناوین کتبی که نوشته اند برقرار میکنیم. در صورتیکه این اطلاعات را به جداول مختلف تقسیم نمیکردیم مجبور بودیم تا اطلاعات نویسنده را برای هر عنوان کتاب در جدول *Titles* در نظر بگیریم و از اینرو یکسری اطلاعات اضافی و تکراری در جدول ذخیره میشد چراکه برای عناوین کتبی که نویسنده آنها یکسان است، اطلاعات نویسنده بطور تکراری ذخیره میگردد.

همچنین برای مقاصد آنالیز و تجزیه و تحلیل، ضروری است تا اطلاعات مختلف از جداول مختلف با یکدیگر ادغام شوند و یک مجموعه اطلاعاتی جدید را ایجاد نمایند. با استفاده از عملی تحت عنوان "*ادغام جداول*"، این عمل قابل اجرا است و بوسیله *INNER JOIN* در دستور *SELECT* آنرا در *SQL* عملی میکنیم. یک *INNER JOIN* رکوردهای مختلفی را از جداول متفاوت با یکدیگر ادغام میکند و این عمل را از طریق تست کردن مقادیر فیلدهایی انجام میدهد که در جداول مورد نظر عمومیت دارند. ساده ترین فرم *INNER JOIN* بشکل زیر است :

```
SELECT fieldName1, fieldName2, ...
FROM table1
INNER JOIN table2
ON table1.fieldName = table2.fieldName
```

که در آن قسمت بعد از عبارت ON، فیلدهایی را نشان میدهد از دو جدول با یکدیگر مقایسه میشوند تا معین شود چه رکوردهایی با یکدیگر ادغام میشوند. برای مثال، query زیر لیستی از تمام نویسندگان را به همراه ISBN مربوط به کتابهای نوشته شده توسط آنها را ایجاد میکند :

```
SELECT firstName, lastName, isbn
FROM Authors
INNER JOIN AuthorISBN
ON Authors.authorID = AuthorISBN.authorID
ORDER BY lastName, firstName
```

این query فیلدهای firstName و lastName از جدول Authors را با فیلد isbn از جدول AuthorISBN ادغام میکند و نتیجه را بر اساس فیلدهای firstName و lastName بصورت صعودی مرتب مینماید. در این query به فرمت عبارت نوشته شده بعد از عبارت ON که بصورت *tableName.fieldName* است توجه نمایید. در این قسمت مشخص میشود که عمل ادغام بین دو جدول از طریق کدام فیلد انجام خواهد گرفت. استفاده از فرمت نوشتاری "*tableName*." در مواردی ضروری است که نام فیلد در هر دو جدول یکسان باشد.

ادغام اطلاعات از جداول *Authors*، *AuthorISBN* و *Publisher* و *Titles*

در اینجا می خواهیم برای پایگاه داده Book یک query تعریف کنیم که توسط آن عنوان کتاب، شماره ISBN، نام نویسنده، نام خانوادگی نویسنده، سال انشار کتاب و نام ناشر هر کتاب نمایش داده شود. برای کتابهایی که بیش از یک نویسنده دارند، این

query میبایست رکوردهای مجزایی را برای هر نویسنده نمایش دهد. توجه نمایید که در این query نیاز است تا بین هر چهار جدول ادغام صورت گیرد. این query در زیر نمایش داده شده است :

```
SELECT      Titles.Title,      Titles.ISBN,      Authors.FirstName,
Authors.LastName, Publishers.PublisherName

FROM

    (

        Publishers

        INNER JOIN Titles

        ON Publishers.PublisherID = Titles.PublisherID

    )

INNER JOIN

    (

        Authors

        INNER JOIN AuthorISBN

        ON Authors.AuthorID = AuthorISBN.AuthorID

    )

ON Titles.ISBN = AuthorISBN.ISBN

ORDER BY Titles.Title;
```

نتیجه این query در شکل زیر نمایش داده شده است :

نتیجه query

حال به بررسی این query میپردازیم. در خطوط ۲ و ۱ این query، فیلدهای مورد نظر که بعنوان نتیجه در خروجی نمایش داده میشوند، قرار گرفته اند. توجه نمایید که ترتیب قرار گرفتن فیلدها در این لیست، از این جهت اهمیت می یابد که فیلدها در خروجی به همین ترتیب نمایش داده خواهند شد. این query، فیلدهای title و isbn از جدول Titles، فیلدهای lastName و firstName از جدول Authors، فیلد copyright از جدول Titles و فیلد publisherName از جدول Publishers را انتخاب مینماید. برای اینکه متن این query خوانا تر باشد، قبل از نام هر فیلد نام جدول آنرا نیز آورده ایم. (به این روش Fully Qualified Name گفته میشود.) در خطهای بعدی، عمل ادغام بین جداول مختلف صورت گرفته است. در این query از سه عمل INNER JOIN استفاده شده است. باید به نکته توجه شود، اگرچه INNER JOIN بر روی دو جدول انجام میشود، اما نتیجه آن می تواند بعنوان ورودی برای INNER JOIN دیگری مورد استفاده قرار گیرد چراکه نتیجه یک INNER JOIN خود یک جدول است. همچنین برای اولویت بخشیدن به اجرای دستورات از پرانتز استفاده نموده ایم. در ابتدا با دستور زیر آغاز میکنیم :

```
(Publishers INNER JOIN Titles
```

```
ON Publishers.publisherID = Titles.publisherID)
```

که جدول Publishers و Titles را با استفاده از شرایطی که در آن فیلدها publisherID دو جدول با یکدیگر منطبق هستند، ادغام میکند. جدول موقتی که از انجام این INNER JOIN تشکیل میشود حاوی اطلاعات کتاب و ناشر مربوط به آن است.

دیگر INNER JOIN این query عبارت است از :

```
( Authors
```

```
INNER JOIN AuthorISBN
```

```
ON Authors.AuthorID = AuthorISBN.AuthorID )
```

که جدول Authors و AuthorISBN را در شرایطی که فیلد AuthorID با یکدیگر منطبق باشند، ادغام مینماید.

سومین INNER JOIN در این query عبارت است از :

```
(
    Publishers
    INNER JOIN Titles
    ON Publishers.PublisherID = Titles.PublisherID
)
INNER JOIN
(
    Authors
    INNER JOIN AuthorISBN
    ON Authors.AuthorID = AuthorISBN.AuthorID
)
ON Titles.ISBN = AuthorISBN.ISBN
```

که باعث ادغام دو جدول موقتی ایجاد شده از دو INNER JOIN قبلی، تحت شرایطی میشود که در آن فیلد Titles.isbn برای هر رکورد در اولین جدول موقت با فیلد AuthorISBN.isbn برای هر رکورد از جدول موقت دوم، منطبق باشند. نتیجه نهایی این INNER JOIN نیز، جدول موقتی است که نتیجه مورد نظر را در بر دارد.

در انتهای query نیز، نتایج بر اساس فیلد Titles.title مرتب میشوند.

دستور INSERT

این دستور باعث وارد کردن رکوردی جدید در جدول میشود. ساده ترین فرم این دستور بشکل زیر است :

```
INSERT INTO tableName (fieldName1, fieldName2, ... )
VALUES (value1, value2, ...)
```

که در آن *tableName* جدولی است که در آن رکورد جدید وارد میشود. به دنبال *tableName* لیستی از فیلدها قرار میگیرد و پس از آن عبارت *VALUES* و به دنبال آن لیستی از مقادیر. توجه نمایید که به ازای هر *fieldName* مقدار *value* در جدول وارد میشود. همچنین باید توجه شود که ترتیب و نوع مقادیر وارد شده در *value* میبایست با نام و نوع *field* مورد نظر همخوانی کامل داشته باشد. برای مثال، اگر *fieldName1* بعنوان *firstName* در نظر گرفته شود، آنگاه مقدار *value1* باید رشته ای باشد که درون یک جفت

' ' قرار گرفته و بیانگر نام باشد. بعنوان یک مثال، *query* زیر را در نظر بگیرید :

```
INSERT INTO Authors (lastName, firstName)
VALUES ('Smith' , 'Sue')
```

این *query* باعث میشود تا رکوردی جدید در جدول *Authors* وارد شود. توجه نمایید در دستور *INSERT* حتما برای فیلدهای *Primary Key* میبایست مقداری وارد گردد، اما در اینجا چون فیلد *Primary Key*، فیلد *authorID* است و این فیلد از نوع *auto-increment* تعریف شده، نیازی به مقداردهی ندارد، چراکه با وارد شده هر رکورد جدید بطور خودکار مقداری برای آن در نظر گرفته میشود. همچنین دقت کنید، بغیر از فیلدهای *Primary Key*، در صورتیکه برای فیلدی از یک جدول در دستور *INSERT* مقداری در نظر گرفته نشود، مقدار آن فیلد تهی (*NULL*) در نظر گرفته خواهد شد.

نکته : *SQL* برای نشان دادن رشته ها از یک جفت ' ' استفاده میکند. برای نشان دادن رشته هایی که خود حاوی ' هستند (مانند *O'Reilly*) باید از فرمت *'O' 'Reily'* استفاده شود.

دستور UPDATE

این دستور باعث ایجاد تغییر در داده های جدول میشود و ساده ترین فرم آن بصورت زیر است :

```
UPDATE tableName
SET fieldName1 = value1, fieldName2 = value2, ...
WHERE criteria
```

که در آن *fieldName* هایی که مقادیر آنها تغییر مییابند مشخص شده و مقادیر جدید (*value*) به آنها تخصیص داده میشود. وجود عبارت WHERE در دستور UPDATE باعث میشود تا رکوردهایی که این فیلدها در آنها تغییر می یابند معین شوند. برای مثال :

```
UPDATE Authors
SET lastName = 'Jones'
WHERE lastName = 'Smith' AND firstName = 'Sue'
```

باعث میشود تا رکوردی که در آن Sue Smith قرار داشته است، به Sue Jones تغییر یابد.

توجه : عدم استفاده از عبارت WHERE پس از دستور UPDATE ممکن است باعث بروز خطایی منطقی گردد.

دستور DELETE

دستور DELETE باعث حذف داده ها از جدول میشود و ساده ترین فرم آن بصورت زیر است :

```
DELETE FROM tableName
WHERE criteria
```

که در آن criteria مشخص کننده رکورد یا رکوردهایی است که از جدول حذف میشوند.

```
DELETE FROM Authors
```

```
WHERE lastName = 'Jones' AND firstName = 'Sue'
```

باعث حذف شدن رکوردی میشود که نام و نام خانوادگی آن Sue Jones است.

نکته : عبارت WHERE ممکن است با چندین رکورد در جدول مورد نظر همخوانی داشته باشد، از اینرو دستور DELETE دقت نماید که شرط نوشته شده در WHERE تنها با رکورد مورد نظر شما همخوانی داشته باشد تا از حذف شدن ناخواسته اطلاعات جلوگیری شود.

مدل شیء ADO.NET (ADO.Net Object Model)

مدل شیء ADO.Net فراهم کننده API ای است که بوسیله آن می توان از طریق کد نویسی به پایگاه داده دسترسی پیدا کرد. ADO.Net مخصوص Net Framework. طراحی شده و نسل جدید ADO میباشد. (کلمه ADO مخفف ActiveX Data Objects است.)

در Net Framework، System.Data فضای است که در آن تمامی API های مربوط به ADO.Net قرار دارند. Namespace های پایه ADO.Net عبارتند از System.Data.OleDb و System.Data.SqlClient که با استفاده از آنها امکان دسترسی و ایجاد تغییر در Data Source های مختلف فراهم میگردد. (منظور از Data Source هر منبعی داده ایست که میتوان در آن اطلاعات و داده ها را ذخیره نمود.)

System.Data.OleDb شامل کلاسهایی است که بوسیله آنها می توان به هر Data Source ای متصل شد، در حالیکه System.Data.SqlClient تنها شامل کلاسهایی جهت استفاده از MS SQL Server 2000 است.

شروع برنامه نویسی با ADO.Net – اتصال به MS Access

برای آغاز کار برنامه نویسی و اتصال به پایگاه داده، با پایگاه داده Access شروع میکنیم که پایگاه داده ای ساده بوده و به همراه بسته نرم افزاری MS Office قابل نصب است. توجه کنید در برنامه زیر، از ساده ترین روش برای اتصال به پایگاه داده استفاده شده و مطالب پیشرفته تر را در آینده بررسی خواهیم نمود.

برای اتصال به پایگاه داده Access میبایست از کلاسهای موجود در System.Data.OleDb استفاده نماییم.

مراحل کار با پایگاه داده بشرح زیر میباشد :

- اتصال به پایگاه داده
- بدست آوردن اطلاعات مورد نظر از پایگاه داده
- نمایش اطلاعات بدست آمده
- قطع ارتباط با پایگاه داده

برای اتصال به پایگاه داده Access از شیء OleDbConnection استفاده میکنیم. توجه کنید که برای اتصال به هر پایگاه داده ای نیاز به یکسری پارامترها و مسیرها است که این اطلاعات در رشته ای تحت عنوان connection string قرار داده میشود و از طریق اطلاعات این رشته به پایگاه داده متصل میشویم. برای اتصال به Access نیز میبایست از یک connection string استفاده نماییم. برای مثال، رشته زیر را در نظر بگیرید :

```
string connectionString =
    "provider=Microsoft.Jet.OLEDB.4.0;" +
    "data source=F:\\Samples\\Books.mdb";
```

این رشته، نمونه ای از یک connection string جهت برقراری ارتباط با پایگاه داده Access است. در قسمت provider نام شرکت پشتیبانی کننده پایگاه داده نوشته میشود که برای Access این مقدار عبارت است از

Microsoft.Jet.OLEDB.4.0 و در قسمت data source محل قرار گرفتن پایگاه داده مشخص میگردد. (محل فیزیکی که در آن پایگاه داده ذخیره شده است).

همچنین به استفاده از "\\\" جهت نمایش "\" توجه نمایید.

در مثال زیر، برنامه نمونه ای مطرح شده که در آن نحوه برقراری ارتباط با پایگاه داده Access از طریق شیء OleDbConnection به نمایش در آمده است.

```
using System;

using System.Data;

using System.Data.OleDb;

class OleDbConnectionAccess
{
    public static void Main()
    {
        // بیان رشته ای که در آن جزئیات ارتباط با پایگاه داده مشخص میشود.

        string connectionString =
            "provider=Microsoft.Jet.OLEDB.4.0;" +
            "data
            source=F:\\Samples\\Books.mdb";

        // ایجاد شیء جدیدی از OleDbConnection جهت برقراری ارتباط با پایگاه
        // که connectionString به سازنده آن ارسال میشود.

        OleDbConnection myOleDbConnection =
            new OleDbConnection(connectionString);
```

```
// OleDbCommand ایجاد شیء
OleDbCommand myOleDbCommand =
myOleDbConnection.CreateCommand();

// SQL دستور OleDbCommand از شیء CommandText قرار دادن ویژگی
// که توسط آن یک سطر از جدول Authors انتخاب میشود.
myOleDbCommand.CommandText =

    "SELECT * "+

    "FROM Authors "+

    "WHERE authorID=1";

// Open () باز کردن ارتباط بین پایگاه داده با استفاده از متد
myOleDbConnection.Open();

// OleDbDataReader و فراخوانی متد ExecuteReader () از
// شیء OleDbCommand جهت اجرای دستور SELECT
OleDbDataReader myOleDbDataReader =
    myOleDbCommand.ExecuteReader();

// OleDbDataReader خواندن یک سطر از
// Read () با استفاده از متد
myOleDbDataReader.Read();
```

```
// نمایش نتیجه و خروجی

Console.WriteLine("myOleDbDataReader\" firstName\" = "+
    myOleDbDataReader["firstName"]);

Console.WriteLine("myOleDbDataReader\" lastName\" = "+
    myOleDbDataReader["lastName"]);

Console.WriteLine("myOleDbDataReader\" AuthorID\" = "+
    myOleDbDataReader["authorID"]);

// بستن شیء OleDbDataReader با استفاده از متد Close()

myOleDbDataReader.Close();

// بستن ارتباط بین پایگاه داده از شیء OleDbConnection

myOleDbConnection.Close();

}

}
```

خروجی این برنامه به شکل زیر است :

```
myOleDbDataReader[" firstName"] = Harvey
myOleDbDataReader[" lastName"] = Deitel
myOleDbDataReader[" AuthorID"] = 1
```

توجه نمایید که در اینجا تنها یک سطر از پایگاه داده فراخوانی شده و با تغییر دستور SQL میتوان نتیجه های دلخواه را بدست آورد.