مقدمه

C# یکی از زبانهای جدید برنامه‌سازی شی‌گرا است که با ارائهٔ رهیافت Component-Based به طراحی و توسعه نرم‌افزار می‌پردازد. آنچه ما در حال حاضر از زبانهای برنامه‌سازی Component-Based در اختیار داریم و آنچه که C# در اختیار ما قرار می‌دهد، افق جدیدی به سوی تولید و طراحی نرم‌افزارهای پیشرفته را در روی ما قرار می‌دهند.

نرم‌افزار، به عنوان یک سرویس، هدف اصلی نسل بعدی در سیستم‌های محاسباتی است. برای مثال، C# زبانی مناسب برای تولید و طراحی صفحات وب، ایجاد اجزایی با قابلیت استفاده مجدد و ایجاد محیط‌هایی چند رسانه‌ای را به عنوان زبانی که هدفش توسعه ایجاد نرم‌افزارهای پیشرفته است، در اختیار ما قرار می‌دهد.

زبان برنامه‌سازی C#، به همراه تکنولوژی جدید شرکت نرم‌افزاری مایکروسافت یعنی.NET ارائه گردید، از این رو از تکنولوژی.NET این شرکت بهره می‌برد. پس در ابتدا به بیان مطالبی درباره محیط.NET می‌پردازیم.

چرا NET.؟

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

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

NET. چیست؟

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

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

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

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

سرویسهای وب، رهیافتی جدید، مستقل از platform و استاندارد، جهت ایجاد ارتباط و فعالیت بین سیستمهای ناهمگون در اینترنت، می‌باشند. سرویسهای وب NET.، از زیر ساخت شی‌گرایی برنامه‌نویسی ASPNET. استفاده می‌کنند، اما همچنان از استانداردهای باز و مدلی بر پایهٔ پیغام (Message Based Model) استفاده می‌نمایند. استفاده از استانداردهای باز از قبیل XML، WSDL و UDDI باعث

به محیط و platform آنها، ارتباط برقرار نمایند.

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

کتابخانه‌های کلاس‌های پایه(Base Class Library: BCL)

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

NET. Namespaces	
Namespace	**Description**
System	The most commonly used types.
System.CodeDom	Allows creating types that automate working with source code, that is, compilers and code creation tools.
System.Collections	Collection types such as ArrayList, Hashtable, and Stack.
System.ComponentModel	Supports building reusable components.
System.Configuration	Types for working with various kinds of XML configuration files.
System.Data	Most of the types for ADONET. database programming. Other types are in namespaces that are specific to a database or data interface.
System.Diagnostics	Process, EventLog, and Performance Counter types.
System.DirectoryServices	Managed interface for accessing Windows Active Directory Services.
System.Drawing	GDI+ types.
System.EnterpriseServices	COM+ types.
System.Globalization	Types for culture-specific support of calendars, formatting, and languages.

.NET Namespaces	
Namespace	**Description**
`System.IO`	`Directory`, `File`, and `Stream` types.
`System.Management`	APIs for performing WMI tasks.
`System.Messaging`	Types for working with message queues.
`SystemNET.`	Access to networking protocol types.
`System.Reflection`	Reflection APIs for inspecting assembly metadata.
`System.Resources`	Types for culture-specific resource management.
`System.Runtime`	COM Interop, Remoting, and Serialization support.
`System.Security`	Code access security, role-based security, and cryptography types.
`System.ServiceProcess`	Types for building Windows Services.
`System.Text`	Text encoding/decoding, byte array from/to string translation, the `StringBuilder` class, and regular expressions.
`System.Timers`	Timer types.
`System.Threading`	Threads and synchronization types.
`System.Web`	HTTP Communications, ASPNET., and Web Services types.
`System.Windows`	Windows Forms types.
`System.XML`	All XML support types, including XML Schema, `XmlTextReaders/XmlTextWriters`, `XPath`, XML Serialization, and XSLT.

جدول ۱-۱ Namespaceهای مهم و رایج

هر Namespace مجموعه‌ای از کلاسهای از پیش ساخته شدهٔNET. است که می‌توان از آنها در برنامه‌های مختلف استفاده نمود.

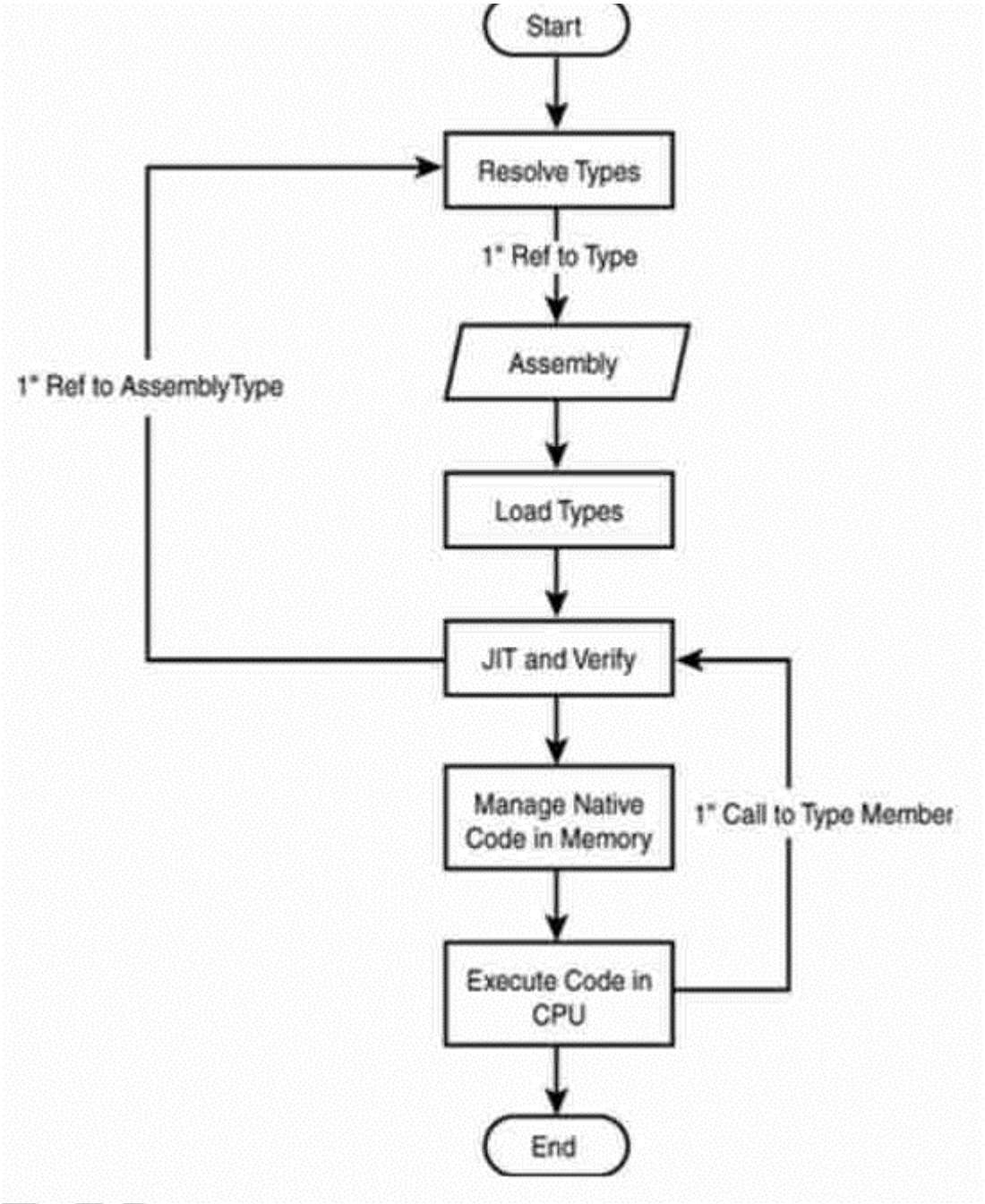
(Common Language Runtime)CLR

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

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

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

مسئلهٔ مهم دیگر در مورد CLR، نحوهٔ بارگذاری(Load) و اجرای برنامه توسط آن است. به محض اینکه برنامهٔ.NET شروع به اجرا می‌کند، ویندوز اسمبلی.NET راتشخیص داده و CLR را اجرا می‌کند. سپس CLR نقطه شروع برنامه را شناسایی و پروسهٔ تعیین انواع که در آن، محل قرارگیری انواع مختلف بکار رفته در برنامه مشخص می‌شود را، اجرا می‌کند. اسمبلی شناسایی شده در پروسهٔ Loader بارگذاری می‌گردد.



شکل ۱-۱ نحوه مدیریت برنامه‌ها توسط CLR

زبانهای برنامه‌نویسی

قسمت مهم دیگرNET.، پشتیبانی آن از چندین زبان برنامه‌نویسی متفاوت است. IL طوری طراحی شده است که از چندین زبان برنامه‌نویسی پشتیبانی نماید. در حقیقت، هم اکنون ده‌ها زبان برنامه‌نویسی مورد پشتیبانی و پذیرش IL می‌باشند. علاوه بر NET.،C# شامل زبانهایی نظیر Visual Basic، Jscript، J# و C++ نیز می‌باشد. برخی دیگر از زبانهای برنامه‌سازی مهم که بوسیلهٔ IL پشتیبانی می‌شوند عبارتند از: Borland DelphiNET.، CobolNET.، Fujitsu.، PythonNET.، PerlNET. و بسیاری دیگر از زبانهای برنامه‌سازی که تحتNET. عمل می‌کنند و بوسیلهٔ آن مورد پذیرش هستند.

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

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

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

مزایای پشتیبانی از چندین زبان برنامه‌نویسی

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

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

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

"خصوصیات عمومی زبان" یا CLS به منظور حل چنین مشکلاتی طراحی شده است.CLS ویژگی‌های عمومی یک زبان را مشخص می‌کند و تعیین می‌کند که زبانها در صورت نیاز به اشتراک‌گذاری کدها تا چه حدی می‌توانند عمل نمایند. بعنوان مثال، C# برای اینکه بخواهد با CLS همخوانی داشته باشد، نباید اشاره‌گرها و نوع‌های بدون علامت را به صورت عمومی(public) در نظر بگیرد.

فصل دوم: 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 ۱-۲، یک برنامهٔ 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!");
 }
 }
```

برنامهٔ لیست ۱-۲ 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# 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 StudioNET. بپردازم. اما فعلاً برای اجرای برنامه‌ها می‌توانید از Visual StudioNET. استفاده کنید. پس از نصب آن، وارد محیط C# شده و در قسمت انتخاب برنامهٔ جدید گزینهٔ Console را جهت اجرای برنامه‌ها انتخاب نمایید.

برای این درس، کلاس با توضیحات بیشتر درباره محیط ویژوال سی‌شارپ.net در آینده با توضیح کامل محیط Visual StudioNET. خواهم پرداخت.

برای اجرای کد بالا در صورتیکه از محیط ویژوال استفاده می‌کنید باید بر روی دکمهٔ 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 StudioNET. (VSNET.) 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.

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

اعلان Namespace به سیستم اعلان می‌نماید که شما از توابع کتابخانه‌ای System جهت اجرای برنامه‌ها خود استفاده می‌نمایید. دستوراتی مانند WriteLine و ReadLine جزو توابع کتابخانه‌ای System می‌باشند. اغلب دستورات و توابع مهم و کلیدی استفاده از کنسول ورودی/خروجی در این کتابخانه می‌باشد. در صورتیکه در ابتدای برنامه از using System استفاده نکنید، باید در ابتدای هر یک از دستورات برنامه که مربوط این کتابخانه است، از کلمهٔ System استفاده نمایید. بعنوان مثال در صورت عدم استفاده از System using 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 معدود عناصر زبان C# است که بوسیلهٔ آن می‌توان به ایجاد یک شی (Object) از قبیل واسط ها (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. کلمهٔ کلیدی Main() که نام متد این کلاس نیز می‌باشد جزو کلمات رزرو شده زبان C# است که مشخص

است. در صورتیکه یک برنامه حاوی متد Main () نباشد بعنوان توابع سیستمی همانند dll های ویندوز در نظر گرفته می‌شود.

قبل از کلمهٔ 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 شیٔ‌ای 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 اولیهٔ برنامه‌نویسی شی‌گرا (OOP) هستند. سعی می‌کنم در درس‌های آینده به توضیح این مفاهیم نیز بپردازم، ولی فعلاً در همین حد کافی می‌باشد.)

هر متد باید دارای یک مقدار بازگشتی باشد، یعنی باید مقداری را به سیستم بازگرداند، در این مثال نوع بازگشتی void تعزیف 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 ادامهٔ 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 ۲-۲ 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 ۲-۲: 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 ۲-۲ 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 گفتم آرایه‌ها را در درس‌های آینده توضیح خواهم داد، هدف از این درس تنها آشنایی با C# است.!)

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 ۲-۲ به آن توجه شود آنست که این برنامه تنها از طریق Command-Lineقابل 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 ۳-۲: 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 می‌شود
  Console.Write("What is your name?: ");
  // متنی نمایش داده شده و برنامه منتظر دریافت ورودی می‌ماند
  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 ۲-۲ اضافه شده است. در اولین خط از این برنامه، متنی با عنوان اینکه نام شما چیست؟ بر روی

کاربر می‌شود. بدین معنی که این بار تا زمانیکه کاربر متنی را به عنوان نام خود وارد نکند اجرای برنامه به پیش نخواهد رفت و خط بعدی اجرا نمی‌شود. این برنامه روش ایجاد ارتباط از طریق برنامه با کاربر را نمایش می‌دهد. در این مثال کاربر دقیقاً متوجه می‌شود که چه زمانی باید اطلاعات را وارد نماید و این اطلاعات چه باید باشد در حالیکه در مثال قبل چنین نبود. همانگونه که می‌بینید در این برنامه آرگومان {0} مستقیماً از طریق دستور Console.ReadLine() دریافت می‌شود و بلافاصله در خروجی نمایش داده می‌شود. دستور ReadLine() نیز یکی از متدهای کلاس Console است که بوسیله آن رشته ورودی خوانده می‌شود.

خروجی برنامه فوق به شکل زیر است:

```
What is your name?:
```

(سپس برنامه منتظر دریافت متنی از ورودی توسط کاربر می‌ماند)

(پس از اینکه کاربر رشته‌ای را وارد کرد و کلیدEnter را فشار داد، متن زیر نمایش داده می‌شود.)

```
Hello, Meysam!
```

(سپس اجرای برنامه به دستور بعدی منتقل می‌شود)

```
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 بازمی‌گرداند. این مقدار در این برنامه به آرگومان {0} برگردانده می‌شود. این خط از برنامه را می‌توان 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 کرده‌ایم که مقدار ورودی در آن ذخیره می‌شود و سپس از این مقدار به عنوان مقداری که {0} می‌پذیرد استفاده کرده‌ایم.

در این درس آموختید که ساختار کلی یک برنامه C# چگونه است. هر برنامه C# از یک کلاس اصلی تشکیل می‌شود که این کلاس شامل داده‌ها و متغیرها و متدهایی می‌باشد. متد آغازین برنامه که برنامه با آن شروع به اجرا می‌کند، متد Main() است. با استفاده از توابع کتابخانه‌ای می‌توان 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 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 اعداد صحیح (integer)، اعداد اعشاری (Floating points) و اعداد دسیمال (Decimal) می‌باشد.(به انواع Boolean از اینرو منطقی می‌گوییم که تنها دارای دو حالت منطقی صحیح (True) و یا غلط (False) می‌باشند.)

مثال 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 یا C++، مقدار عددی نمی‌پذیرند، زیرا همانگونه که می‌دانید در این دو زبان هر مقدار عددی صحیح مثبت بغیر از صفر به عنوان true و عدد صفر به عنوان

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.

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

از این انواع برای محاسبات عددی استفاده می‌گردد. یک نوع دیگر را نیز می‌توان در این جدول اضافه نمود و آن نوع char است. هر چند شاید از نظر بسیاری از دوستانی که با زبانهای دیگر برنامه‌سازی کار کرده‌اند این تقسیم بندی غلط به نظر آید، اما باید گفت که در زبان C# نوع char نیز نوع خاصی از انواع عددی است که رنجی بین صفر تا ۶۵۵۳۵ دارد و اندازه آن نیز ۱۶ بیتی است، اما به جای نمایش دادن مقادیر عددی تنها می‌تواند بیان کننده یک کاراکتر باشد. در آینده در این مورد بیشتر توضیح خواهم داد.

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

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

انواعی از نوع floating point هنگامی استفاده می‌شوند که محاسبات عددی به دقت‌های اعشاری نیاز داشته باشند. همچنین برای منظورهای تجاری استفاده از نوع decimal بهترین گزینه

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 شرکت‌پذیری آنها را نشان می‌دهد.

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

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 = (sbyte)(~bitNot);
Console.WriteLine("Bitwise Not: {0}", bitNot);
logNot = false;
logNot = !logNot;
Console.WriteLine("Logical Not: {0}", logNot);
}
}
```

به هنگام محاسبه عبارات، دو عملگر x++ و x—(که در اینجا کاراکتر x بیان کننده آن است که عملگرهای ++ و – در جلوی عملوند قرار می‌گیرند post-increment و post-decrement) 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 و 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

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 مکمل‌گیری) متغیر bitNot بعنوان یک بایت در نظر گرفته می‌شود و مقدار آن منفی یا نقیض می‌شود. در عملیات بیتی نقیض بدین معناست که تمامی یکها به صفر و تمامی صفرها به یک تبدیل شوند. در این حالت نمایش باینری عدد صفر یا همان 00000000 به نقیض آن یعنی 11111111 تبدیل می‌گردد.

در این مثال به عبارت (sbyte) (~bitNot) توجه نمایید. هر عملی که بر روی انواع short،unshort، byte و sbyte انجام شود، مقداری از نوع int را باز می‌گرداند. بمنظور اینکه بتوانیم نتیجه دلخواه را به متغیر bitNot تخصیص دهیم باید از فرمت (Type) operator استفاده نماییم که در آن Type نوعی است می‌خواهیم نتیجه ما به آن نوع تبدیل شود و operator عملی است که بر روی متغیر صورت می‌پذیرد. به بیان دیگر چون می‌خواهیم مقدار متغیر bitNot بصورت بیتی در نظر گرفته شود، پس باید نتیجه عمل ما بصورت بیتی در آن ذخیره شود که استفاده از نوع sbyte باعث می‌شود تا نتیجه به فرم بیتی (یا بایتی) در متغیر ما ذخیره شود. باید توجه نمایید که استفاده از فرمت (Type) یا در اصطلاح عمل Casting، در مواقعی که می‌خواهیم تغییری از یک نوع بزرگتر به نوع کوچکتر ایجاد نماییم، مورد استفاده قرار گیرد، چرا که در این حالات ممکن است با از دست دادن اطلاعات مواجه باشیم. در این مثال چون می‌خواهیم نوع بزرگتر int را به(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 داده‌های شما را تهدید می‌نماید. بحث در مورد مسائل پیچیده‌تر در مورد تبدیل انواع علامتدار و و بدون علامت به یکدیگر در اینجا نمی‌گنجد و سعی می‌کنم تا آنها را در مطالب بعدی و در جای لازم مورد

نگران نباشید چراکه در آینده در مثالهایی که خواهید دید تمامی این مطالب را در عمل نیز حس کرده و با آنها آشنا خواهید شد.)

عملگر بعدی که در این برنامه مورد استفاده قرار گرفته است، عملگر نقیض منطقی یا همان " ! " است که امکان تغییر مقدار یک متغیر `Boolean` را از `true` به `false` و بالعکس را فراهم می‌آورد. در مثال بالا(مثال شماره 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: 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 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 تعدا 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 صرفه‌جویی حافظه بسیار موثر میباسند. 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

موارد 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، تعدا 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 ذخیره‌سازی داده‌های ما می‌باشد که با دادن این شماره می‌توانیم به داده مورد نظر دسترسی پیدا کنیم. در C# همانند C و C++ اندیس خانه‌های آرایه از صفر آغاز می‌گردد یعنی اولین خانه آرایه دارای شماره صفر است و عناصر بعدی به ترتیب یک واحد به اندیسشان اضافه می‌گردد. پس شماره اندیس آرایه همیشه یک واحد کمتر از تعداد عناصر آن است، یعنی آرایه‌ای که ۱۰ عضو دارد بزرگترین اندیس خانه‌هایش ۹ می‌باشد. 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 می‌گویند. مثلاً `monthArray[3,7]` عنصر واقع در سطر ۳ و ستون ۷ آرایه را مشخص می‌نماید.(توجه داشته باشید که اندیس دهی آرایه از صفر آغاز می‌شود و بعنوان مثال `intArray[12]` به خانه شماره ۱۲ آرایه اشاره می‌کند اما فراموش نکنید چون اندیس آرایه از صفر آغاز می‌شود خانه شماره ۱۲ آرایه، سیزدهمین داده شما را در خود جای می‌دهد.)

۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲

اگر شکل فوق را آرایه‌ای تک بعدی تصور نمایید، مشاهده می‌نمایید که خانه شماره ۵ آرایه حاوی اطلاعات مربوط به ششمین داده ورودی شما می‌باشد.

نکته دیگری که باید در مورد تعریف آرایه‌های این مثال متذکر شوم در مورد آریه‌هائیست که از نوع string تعریف می‌شوند. دوستانی که با زبان C کار کرده‌اند حتماً می‌دانند که آرایه‌ای از نوع رشته‌ای 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 نک بعدی محلی برای ذخیره‌سازی یک رشته است و همانند زبان C نیاز به پردازش‌های گاه پیچیده بر روی آرایه‌های چند بعدی بمنظور کار با رشته‌ها، وجود ندارد. بعنوان یک توضیح کمی اختصاصی عرض می‌کنم که در زبان‌هایی مانند C، در صورتیکه می‌خواستید چندین رشته را در آرایه‌ای 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 می‌توانستید به عناصر رشته‌ای آرایه دسترسی پیدا کنید، اما در C# تنها با استفاده از یک آرایه تک بعدی می‌توان 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 می‌کنید، آرایه‌ای از نوع رشته تعریف شده و دو عنصر به آن تخصیص داده شده است و در جایی در متن برنامه با استفاده از اندیس از اولین عنصر این آرایه برای نمایش در خروجی استفاده گردیده است. خروجی این برنامه به شکل زیر است:

```
My name is Meysam
```

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

فصل چهارم 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 ۱-۴ – فرم‌های دستورالعمل if

```
using System;

class IfSelect
{
 public static void Main()
 {
 string myInput;
 int myInt;
 Console.Write("Please enter a number: ");
 myInput = Console.ReadLine();
 myInt = Int32.Parse(myInput);
// تصمیم‌گیری ساده و اجرای عمل داخل دو کروشه
 if (myInt > 0)
  {
  Console.WriteLine("Your number {0} is greater than zero.", myInt);
  }
//تصمیم‌گیری ساده و اجرای عمل بدون استفاده از دو کروشه
 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);
  }
```

```
 {
 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 ۱-۴ از یک متغیر myInt برای دریافت ورودی از کاربر استفاده می‌نماید، سپس با استفاده از یک سری دستورات کنترلی، که همان دستور if در اینجاست، عملیات خاصی را بسته به نوع ورودی انجام می‌دهد. در ابتدای این برنامه عبارت Please enter a umber: در خروجی چاپ می‌شود. دستور Console.ReadLine() منتظر می‌ماند تا کاربر ورودی وارد کرده و سپس کلید Enter را فشار دهد. همانطور که در قبل نیز اشاره کرده‌ایم، دستور Console.ReadLine() عبارت ورودی را به فرم رشته دریافت می‌نماید پس مقدار ورودی کاربر در اینجا که یک عدد است به فرم رشته‌ای در متغیر myInput که از نوع رشته‌ای تعریف شده است قرار می‌گیرد. اما میدانیم که برای اجرای محاسبات و یا تصمیم‌گیری بر روی اعداد نمی‌توان از آنها در فرم رشته‌ای استفاده کرد و باید آنها را بصورت عددی مورد استفاده قرار داد. به همین منظور باید متغیر myInput را به نحوی به مقدار عددی تبدیل نماییم. برای این منظور از عبارت Int32.Parse() استفاده می‌نماییم. این دستور مقدار رشته‌ای متغیر داخل پرانتزش را به مقدار عددی تبدیل کرده و آنرا به متغیر دیگری از نوع عددی تخصیص می‌دهد. در این مثال نیز همانطور که دیده می‌شود، myInput که تز نوع رشته‌ای است در داخل پرانتز قرار گرفته و این مقدار برابر با myInt که از نوع int است قرار گرفته است. با این کار مقدار عددی رشته ورودی کاربر به متغیر myInt تخصیص داده می‌شود. (توضیح کامل‌تری در مورد Int32 و سایر تبدیلات مشابه به آن در درسهای آینده و در قسمت

استفاده از دستور if بر روی آن پردازش انجام داده و تصمیم‌گیری نماییم.

دستور if

اولین دستور بصورت if (boolean expression) {statements} آورده شده است. دستور if با استفاده از کلمه کلیدی if آغاز می‌شود. سپس یک عبارت منطقی درون یک زوج پرانتز قرار می‌گیرد. پس از بررسی این عبارات منطقی دستورالعمل/دستورالعمل‌های داخل کروشه اجرا می‌شوند. همانطور که مشاهده می‌نمایید، دستور if یک عبارت منطقی را بررسی می‌کند. در صورتیکه مقدار این عبارات true باشد دستورهای داخل بلوک خود را اجرا می‌نماید(قبلا توضیح داده شد که دستورهایی که داخل یک زوج کروشه {} قرار می‌گیرند در اصطلاح یک بلوک نامیده می‌شوند.) و در صورتیکه مقدار آن برابر با false باشد اجرای برنامه به بعد از بلوک if منتقل می‌شود. در این مثال همانطور که ملاحظه می‌نمایید، عبارت منطقی دستور if 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 دراین 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 درصورتیکه 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 درغیر 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

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 <= 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

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 ۲-۴ – دستورالعمل switch

```
using System;

class SwitchSelect
 {
 public static void Main()
 {
 string myInput;
 int myInt;
 begin:
 Console.Write("Please enter a number between 1 and 3: ");
 myInput = Console.ReadLine();
 myInt = Int32.Parse(myInput);
 // دستور switch بهمراه 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.Write("Type \"continue\" to go on or \"quit\" to stop: ");
 myInput = Console.ReadLine();
 //دستور switch بهمراه متغیری از نوع رشته‌ای
 switch (myInput)
 {
 case "continue":
  goto begin;
 case "quit":
  Console.WriteLine("Bye.");
  break;
 default:
```

```
  goto decide;
 } //switch پایان بلوک
 } //Main() پایان متد
} //SwitchSelect پایان کلاس
```

مثال ۲-۴ دو مورد استفاده از دستور switch را نشان می‌دهد. دستور switch بوسیله کلمه کلیدی switch آغاز شده و به دنبال آن عبارت دستور switch قرار می‌گیرد. عبارت دستور switch می‌تواند یکی از انواع زیر باشد: sbyte, byte, short, ushort, int, uint, long, ulong, char, string, enum (نوع enum در مبحث جداگانه‌ای مورد بررسی قرار خواهد گرفت.) در اولین دستور switch در مثال ۲-۴، 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 درصورتیکه 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 دستورالعملی 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 ۲-۴، 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 جائیکه 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

می‌خواهد اجرای برنامه را ادامه دهد یا خیر.(با وارد کردن گزینه‌های continue یا quit) همانطور که می‌بینید در اینجا نیز حلقه‌ای تولید شده است.

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

نکته پایانی این مبحث آنست که توجه نمایید که به جای استفاده از دستور switch می‌توانید از چندین دستور if-else استفاده 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 دستور شرطی if آشنا شدید. همچنین با دستور switch و استفاده دستور goto نیز آشنایی پیدا کردید. در پایان مجدداً یادآوری می‌کنم که در استفاده از دستور goto با احتیاط عمل نمایید و به جز در موارد ضروری از آن استفاده نکنید.

در این درس نحوه استفاده از دستورالعمل‌های 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 دستورالعمل break
- ✓ فراگیری نحوه بکارگیری دستورالعمل continue

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

حلقه while

ابتدا به مثال زیر توجه نمایید.

```
using System;

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

مثال ۱-۵ که در بالا ملاحظه می‌کنید، یک حلقه while ساده را نشان می‌دهد. این حلقه با کلمه کلیدی while آغاز شده و سپس به دنبال آن یک عبارت منطقی قرار می‌گیرد و مورد بررسی قرار می‌گیرد. تمامی دستورالعمل‌های کنترلی از یک عبارت منطقی بهره می‌گیرند و این بدین معناست که ابتدا این عبارت باید بررسی شود تا مشخص شود مقدار این عبارت true است یا false. در این مثال مقدار متغیر myInt مورد بررسی قرار می‌گیرد تا چک شود آیا مقدارش از 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.

خروج بلوک while ابتدا مقدار متغیر مقایسه در شرط بررسی می‌شود و سپس یک واحد به مقدار این متغیر افزوده می‌گردد. پس از اتمام بلوک while، عبارت منطقی مجددا کنترل می‌شود و در صورتیکه این عبارت مقدار true بازگرداند، حلقه while مجدداً اجرا می‌شود. زمانیکه عبارت منطقی مقدار false برگرداند، اجرا برنامه به اولین دستور بعد از بلوک while منتقل می‌شود. در این مثال اعداد صفر تا 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 ۲-۵ 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":
```

```
      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 ۲-۵ 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 نمونه‌هائیست که تا کنون با آنها آشنا شدیم و یکی از دو مقدار true یا false را بر می‌گرداند.

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

یک نگاه کلی به مثال ۲-۵ بیندازیم. در متد Main() متغیر myChoice را از نوع رشته‌ای تعریف نموده‌ایم. سپس یکسری دستورات را بر روی کنسول چاپ نموده‌ایم. این دستورات منوهای انتخاب برای کاربر هستند. ما باید ورودی از کاربر دریافت کنیم که چون این عمل از طریق Console.ReadLine() صورت گرفته، باید در متغیری از نوع رشته‌ای قرار گیرد و از اینرو این ورودی را در myChoice قرار داده‌ایم. ما باید ورودی را از کاربر دریافت کنیم و بر روی آن پردازش انجام دهیم. یک روش کارآمد برای این منظور استفاده از دستورالعمل switch است. همانطور که در دستور switch ملاحظه می‌کنید، بری default نیز دستوری در نظر گرفته شده است که نشان می‌دهد مقدار ورودی معتبر نیست.

حلقه for

به مثال ۳-۵ 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 ۳-۵ یک حلقه for را نشان می‌دهد. استفاده از حلقه for برای زمانی مناسب است که دقیقاً بدانید که حلقه چندبار باید تکرا شود. محتویات درون پرانتزهای حلقه for از سه قسمت تشکیل شده است:

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

initializer list لیستی از عبارات است که بوسیله کاما از یکدیگر جدا می‌شوند. این عبارات تنها یکبار در طول دوره کاری حلقه for پردازش می‌شوند. همانطور که در مثال ۳-۴ نیز ملاحظه می‌کنید، این قسمت معمولا 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 معمولا جهت کنترل متغیر شمارشی استفاده می‌شود.

هنگامیکه عبارت منطقی مقدار true بازگرداند، دستورالعمل‌های بلوک for اجرا می‌شوند. در مثال ۳-۴ ما از دو دستور if درون حلقه for نیز استفاده کرده‌ایم. اولین دستور if بررسی می‌کند که آیا مقدار متغیر i برابر با ۱۰ هست یا نه. در اینجا یک نمونه دیگر از استفاده دستور break را ملاحظه می‌کنید. عملکرد دستور break در اینجا نیز همانند مورد استفاده آن در دستور switch است. در صورت اجرای دستور break اجرای حلقه for خاتمه یافته و اجرای برنامه به اولین دستور بعد از حلقه for منتقل می‌شود.

دومین دستور if با اسقتاده از عملگر باقیمانده (%) بررسی می‌کند که آیا متغیر i بر ۲ بخش پذیر هست یا نه. در صورتیکه متغیر i بر ۲ بخش پذیر باشد، دستور continue اجرا می‌شود. پس از اجرای دستور continue از سایر دستورات حلقه for که بعد از continue قرار گرفته‌اند صرفه‌نظر می‌شود و اجرای برنامه به اول حلقه for باز می‌گردد.

قسمت سوم در حلقه for، قسمت postloopaction list است. پس از اینکه تمامی دستورات درون حلقه for اجرا شد، اجرای حلقه به این قسمت باز می‌گردد. این قسمت لیستی از عملیاتی است که می‌خواهید پس از اجرای دستورات درون بلوک حلقه for انجام شوند. در مثال ۳-۵ 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 زمانیکه عبارت منطقی برابر با true باشد اجرا می‌شود.

حلقه foreach

به مثال ۴-۵ 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 ۴-۵ نیز مورد استفاده قرار گرفته است. اولین کاری که در متد Main() صورت گرفته، اعلان آرایه names از نوع رشته‌ای و مقدار دهی آن است.

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

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

می‌شود، تعریف کرده‌ایم. تا زمانیکه اسمی در آرایه names وجود داشته باشد، در متغیر person قرار می‌گیرد و درون حلقه foreach بوسیله دستور Console.WriteLine() در خروجی نمایش داده می‌شود.

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

فصل ششم 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 تقیسم 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 دستورالعمهای متد، دستورهایی از زبان C# هستند که بوسیله آنها عمل مورد نظر برنامه‌نویس انجام می‌شود و عملی است که یک متد آنرا انجام می‌دهد. مثال ۱-۶ 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 می‌شود
Console.WriteLine();
Console.Write("Press Enter key to continue...");
Console.ReadLine();
Console.WriteLine();
} while (myChoice != "Q" && myChoice != "q");
// اجرای برنامه تا زمانیکه کاربر بخواهد ادامه می‌یابد
}
string getChoice()
{
 string myChoice;
 // منویی را نمایش می‌دهد
 Console.WriteLine("My Address Book\n");
 Console.WriteLine("A - Add New Address");
 Console.WriteLine("D - Delete Address");
 Console.WriteLine("M - Modify Address");
 Console.WriteLine("V - View Addresses");
 Console.WriteLine("Q - Quit\n");
 Console.Write("Choice (A,D,M,V,or Q): ");
 // ورودی دریافتی از کاربر را بررسی می‌کند
 myChoice = Console.ReadLine();
 Console.WriteLine();
 return myChoice;
```

}

برنامه مثال ۱-۶ دقیقا 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، پرانتزهای 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 شیء‌ای 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.

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 ۲-۶ توجه کنید.

```
using System;

class Address
{
  public string name;
  public string address;
}//Address پایان کلاس
class MethodParams
{
  public static void Main()
    {
      string myChoice;
      MethodParams mp = new MethodParams();
      do
        {
          // منویی نمایش داده شده و ورودی از کاربر دریافت می‌گردد
          myChoice = mp.getChoice();
          // تصمیمی بر اساس ورودی کاربر گرفته می‌شود
          mp.makeDecision(myChoice);
          // جهت دیدن نتایج توسط کاربر، اجرای برنامه موقتا موقف می‌گردد
          Console.Write("Press Enter key to continue...");
          Console.ReadLine();
          Console.WriteLine();
   } while (myChoice != "Q" && myChoice != "q");
       // اجرای حلقه تا زمانیکه کاربر بخواهد ادامه پیدا می‌نماید
    }//Main پایان متد
  // نمایش منو و دریافت ورودی از کاربر
  string getChoice()
    {
      string myChoice;
      // نمایش منو
      Console.WriteLine("My Address Book\n");
      Console.WriteLine("A - Add New Address");
      Console.WriteLine("D - Delete Address");
      Console.WriteLine("M - Modify Address");
      Console.WriteLine("V - View Addresses");
```

```
    Console.WriteLine("Choice (A,D,M,V,or Q): ");
    // دریافت ورودی کاربر
    myChoice = Console.ReadLine();
    return myChoice;
  }//getChoice() پایان متد
// 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);
  }
```

```
  void deleteAddress(string name)
    {
      Console.WriteLine("You wish to delete {0}'s address.", name);
    }
  // تغییر یک آدرس
  void modifyAddress(out Address addr)
    {
      //Console.WriteLine("Name: {0}.", addr.name); // خطا رخ می‌دهد
      addr = new Address();
      addr.name = "Meysam";
      addr.address = "C# Persian";
    }
  // نمایش آدرس‌ها
  void viewAddresses(params string[] names)
    {
      foreach (string name in names)
        {
          Console.WriteLine("Name: {0}", name);
        }
    }
}
```

مثال ۲-۶، نمونه تغییر یافته مثال ۱-۶ 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 ۲-۶ 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.

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 فراخوانده آن بازگشت داده می‌شوند. از آنجائیکه این پارامترها از متد فراخوانده 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.

فصل 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)

- ✓ 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()
  {
    // OutputClass نمونه‌ای از
```

```
class.");
    // Output فراخوانی متد کلاس
   outCl.printString();
  }
}
```

در مثال ۱-۷ 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 سازنده‌هاییی 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 ۱-۷ قرار داشت. این سازنده پیش فرض به یک تخصیص‌دهنده همراه شده است. ":" ابتدای تخصیص‌دهنده را مشخص می‌نماید، و به دنبال آن کلمه کلیدی this آمده است. کلمه کلیدی this به شیء کنونی اشاره می‌نماید. استفاده از این کلمه، فراخوانی به سازنده شیء کنونی که در آن تعریف شده است، ایجاد می‌کند. بعد از کلمه کلیدی this لیست پارامترها قرار می‌گیرد که در اینجا یک رشته است. عملی که تخصیص‌دهنده فوق انجام می‌دهد، باعث می‌شود تا سازنده OutPutClass رشته‌ای را بعنوان آرگومان دریافت نماید. استفاده از تخصیص‌دهنده‌ها تضمین می‌نمایند که فیلدهای کلاس شما در هنگام ایجاد نمونه‌ای جدید مقداردهی می‌شوند.

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

در زبان C#، اعضای کلاسها دو نوع می‌باشند: اعضای نمونه و استاتیک. اعضای نمونه کلاس متعلق به رخداد خاصی از کلاس هستند. هربار که شیای 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");
```

myString و متد printString() هستند و این فیلدها و متدها کاملاً از یکدیگر مجزا می‌باشند. به بیان دیگر درصورتیکه عضوی از کلاس استاتیک باشد از طریق ساختار نوشتاری <class name>.<static class member> 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 امکاناتNET. 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:

- 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.

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.
```

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 توانائیهای ParentClass را دارد. از اینرو می‌توان گفت ChildClass یک ParentClass است. (ChildClass **IS** a ParentClass) ChildClass 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 ۱-۸ توجه نمایید. سازنده ParentClass قبل از سازنده ChildClass اجرا می‌گردد.

برقراری ارتباط کلاس مشتق شده با کلاس پایه

به مثال ۲-۸ 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 بهمراه 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 سطرها سوم و چهارم خروجی دیده می‌شوند.

روش دیگر دسترسی به اعضای کلاش 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

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. درصورتیکه از کلمه کلیدی new استفاده نشود، کامپایلر پیغام اخطاری 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.

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 شیگرایی، چند ریختی (Polymorphism) است. با استفاده از این ویژگی، می‌توان برای متد کلاس مشتق شده پیاده سازی متفاوتی از پیاده سازی متد کلاس پایه ایجاد نمود. این ویژگی در جایی مناسب است که میخواهید گروهی از اشیاء را به یك آرایه تخصیص دهید و سپس از متد هر یك از آنها را استفاده کنید. این اشیاء الزاما نباید از یك نوع شیء باشند. هرچند اگر این اشیاء بواسطه ارثبری به یکدیگر مرتبت باشند، می‌توان آنها را بعنوان انواع ارثبری شده به آرایه اضافه نمود. اگر هر یك از این اشیاء دارای متدی با نام مشترك باشند، آنگاه می‌توان هر یك از آنها را جداگانه پیاده سازی 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 یك 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 نمایید 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 ۲-۹، سه کلاس دیده مي شود. این کلاسها از کلاس DrawingObject ارثبري مي‌كنند. هر يك از اين كلاسها داراي متد Draw() هستند و تمامي آنها داراي پيشوند override مي‌باشند. وجود كلمه كليدي override قبل از نام متد، اين امكان را فراهم مي‌نمايد تا كلاس، متد كلاس پايه خود را override كرده و آنرا به طرز دلخواه پيادەسازي نمايد. متدهاي override شده بايد داراي نوع و پارامترهاي مشابه متد كلاس پايه باشند.

پيادەسازي چند ريختي

```
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;
  }
}
```

و ۲-۹ 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 Line، Circle 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 می کند. 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 می شوند. خروجی این برنامه بصورت زیر است:

I'm a Line.
I'm a Circle.
I'm a Square.
I'm just a generic drawing object.

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

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

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

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

مثال ۱-۱۰: یک نمونه از چگونگی دسترسی به فیلدهای کلاس به طریقه 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 میدهد. کلاس PropertyHolder دارای فیلدی است تمایل داریم به آن دسترسی داشته باشیم. این کلاس دارای دو متد getSomeProperty() و setSomePropery() میباشد. متد getSomeProperty()

setSomeProperty() مقداري را به فيلد someProperty تخصيص مي‌دهد.

كلاس PropertyTester از متدهاي كلاس PropertyHolder جهت دريافت مقدار فيلد someProperty از كلاس PropertyHolder استفاده مي‌كند. در متد Main() نمونه جديدي از شي PropertyHolder با نام propHold ايجاد مي‌گردد. سپس بوسيله متد setSomeProperty، مقدار someMethod از propHold برابر با ۵ مي‌گردد و سپس برنامه مقدار property را با استفاده از فراخواني متد Console.WriteLine() در خروجي نمايش مي‌دهد. آرگومان مورد استفاده براي بدست آوردن مقدار property فراخواني به متد getSomeProperty() است كه توسط آن عبارت "Property Value: 5" در خروجي نمايش داده مي‌شود.

چنين متد دسترسي به اطلاعات فيلد بسيار خوب است چرا كه از نظريه كپسوله كردن شيءگرايي پشتيباني مي‌كند. اگر پياده‌سازي someProperty نيز تغيير يابد و مثلا از حالت int به byte تغيير يابد، باز هم اين متد كار خواهد كرد. حال همين مسئله با استفاده از خواص Property ها بسيار ساده‌تر پياده‌سازي مي‌گردد. به مثال زير توجه نماييد.

مثال ۲-۱۰: دسترسي به فيلدهاي كلاس 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);
```

```
}
}
```

مثال ۲-۱۰ چگونگي ايجاد و استفاده از ويژگيها (Property) را نشان مي‌دهد. كلاس PropertyHolder داراي پياده‌سازي از ويژگي SomeProperty است. توجه نماييد كه اوليد حرف از نام ويژگي با حرف بزرگ نوشته شده و اين تنها تفاوت ميان اسم ويژگي SomeProperty و فيلد someProperty مي‌باشد. ويژگي داراي دو accessor با نامهاي set و get است. get accessor مقدار فيلد someProperty را باز مي‌گرداند. set accessor نيز با استفاده از مقدار value، مقداري را به someProperty تخصيص مي‌دهد. كلمه value كه در set accessor آورده شده است جزو كلمات رزرو شده زبان C# مي‌باشد.

كلاس PropertyTester از ويژگي someProperty مربوط به كلاس PropertyHolder استفاده مي‌كند. اولين خط در متد Main() شيء از نوع PropertyHolder با نام propHold ايجاد مي‌نمايد. سپس مقدار فيلد someProperty مربوط به شيء propHold، با استفاده از ويژگي SomeProperty به ۵ تغيير مي‌يابد و ملاحظه مي‌نماييد كه مسئله به همين سادگي است و تنها كافي است تا مقدار مورد نظر را به ويژگي تخصيص دهيم.

پس از آن، متد Console.WriteLine() مقدار فيلد someProperty شيء propHold را چاپ مي‌نمايد. اين عمل با استفاده از ويژگي SomeProperty شيء propHold صورت مي‌گيرد.

ويژگيها را مي‌توان طوري ايجاد نمود كه فقط خواندني (Read-Only) باشند. براي اين منظور تنها كافيست تا در ويژگي فقط از get accessor استفاده نماييم. به مثال زير توجه نماييد.

ويژگيهاي فقط خواندني (Read-Only Properties)

مثال ۳-۱۰: ويژگيهاي فقط خواندني

```
using System;
public class PropertyHolder
{
  private int someProperty = 0;
  public PropertyHolder(int propVal)
  {
    someProperty = propVal;
  }
  public int SomeProperty
  {
    get
    {
```

```
    }
  }
}
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 چگونگي ايجاد يك ويژگي فقط خواندني را نشان مي‌دهد. كلاس PropertyHolder داراي ويژگي SomeProperty است كه فقط get accessor را پياده‌سازي مي‌كند. اين كلاس PropertyHolder داراي سازنده‌ايست كه پارامتري از نوع int دريافت مي‌نمايد.

متد Main() از كلاس PropertyTester شيء جديدي از PropertyHolder با نام propHold ايجاد مي‌نمايد. اين نمونه از كلاس PropertyHolder از سازندة آن كه مقداري صحيح را بعنوان پارامتر دريافت مي‌كند، استفاده مي‌كند. در اين مثال اين مقدار برابر با ۵ در نظر گرفته مي‌شود. اين امر باعث تخصيص داده شدن عدد ۵ به فيلد someProperty از شيء propHold مي‌شود.

تا زمانيكه ويژگي SomeProperty از كلاس PropertyHolder فقط خواندني است، هيچ راهي براي تغيير مقدار فيلد someProperty وجود ندارد. بعنوان مثال در صورتيكه عبارت propHold.SomeProperty = 7 را در كد برنامه اضافه نماييد، برنامة شما كامپايل نخواهد شد چراكه ويژگي SomeProperty فقط خواندني است. اما اگر از اين ويژگي در متد Console.WriteLine() استفاده نماييد بخوبي كار خواهد كرد زيرا اين دستور تنها يك فرآيند خواندن است و با استفاده از get accessor اين عمل قابل اجرا است.

ويژگيهاي فقط نوشتني (Write-Only Properties)

به مثال زير توجه فرماييد:

مثال ۴-۱۰: ويژگيهاي فقط خواندني

```
using System;
public class PropertyHolder
{
  private int someProperty = 0;
  public int SomeProperty
  {
    set
```

```
    somePropery = 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 چگونگي ايجاد و استفاده از ويژگي فقط نوشتني را نشان مي‌دهد. در اين حالت get accessor را از ويژگي SomeProperty حذف كرده و به جاي آن set accessor را قرار داده ايم.

متد Main() كلاس PropertyTester شيء جديد از همين كلاس با سازندة پيش فرض آن ايجاد مي‌نمايد. سپس با استفاده از ويژگي SomeProperty از شيء propHold، مقدار ۵ را به فيلد someProperty مربوط به شيء propHold تخصيص مي‌دهد. در اين حالت set accessor مربوط به ويژگي SomeProperty فراخواني شده و مقدار ۵ را به فيلد someProperty تخصيص مي‌دهد و سپس عبارت “someProperty is equal to 5” را در خروجي نمايش مي‌دهد.

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

- یك struct یا ساختار (Structure) چیست؟
- پیاده‌سازي ساختارها(Struct)
- استفاده از ساختارها(Struct)
- نکات مهم و مطالب کمکي دربارة 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 مقداري (value types) جدید و مورد نظر را ایجاد نمود. از آنجائیکه struct ها بعنوان انواع مقداري در نظر گرفته مي شوند، از اینرو تمامي اعمال مورد استفاده بر روي انواع مقداري را میتوان براي struct ها در نظر گرفت. struct ها بسیار شبیه به کلاس ها هستند و میتوانند داراي فیلد، متد و property باشند. عموماً ساختارها مجموعه کوچکي از عناصري هستند که منطقي با یکدیگر داراي رابطه میباشند. براي نمونه میتوان به ساختار Point موجود در Framework SDK اشاره کرد که حاوي دو property با نامهاي X و Y است.

با استفاده از ساختارها (struct) میتوان اشیایي با انواع جدید ایجاد کرد که این اشیاء میتوانند شبیه به انواع موجود (int, float, ...) باشند. حال سوال اینست که چه زماني از ساختارها(struct) بجاي کلاس استفاده میکنیم؟ در ابتدا به نحوه استفاده از انواع موجود در زبان C# توجه نمایید. این انواع داراي مقادیر و عملگرهاي معیني جهت کار با این مقادیر هستند. حال اگر نیاز به شيای دارید که همانند این انواع رفتار نمایند لازم است تا از ساختارها (struct) استفاده نمایید. در ادامه این مبحث نکات و قوانیني را ذکر میکنیم که با استفاده از آنها بهتر بتوانید از ساختارها (struct) استفاده نمایید.

اعلان و پیاده‌سازي struct

براي اعلان یك struct کافیست تا با استفاده از کلمه کلیدي struct که بدنبال آن نام مورد نظر براي ساختار آمده استفاده کرد. بدنة ساختار نیز بین دو کروشة باز و بسته {} قرار خواهد گرفت. به مثال زیر توجه نمایید:

مثال ۱-۱۱: نمونه‌اي از یك ساختار (Struct)

```
using System;
struct Point
```

```
  public int x;
  public int y;
  public Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  public Point Add(Point pt)
  {
    Point newPt;
    newPt.x = x + pt.x;
    newPt.y = y + pt.y;
    return newPt;
  }
}
/// <summary>
/// struct مثالي از اعلان و ساخت يك
/// </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 ۱-۱۱ نحوة ايجاد و استفاده از struct را نشان مي‌دهد. به راحتي مي‌توان گفت كه يك نوع(type)، يك struct است، زيرا از كلمه كليدي struct در اعلان خود بهره مي‌گيرد. ساختار پايه‌اي يك ساختار پايه‌اي يك struct بسيار شبيه به يك كلاس است، ولي تفاوتهايي با آن دارد كه اين تفاوتها در پاراگراف بعدي مورد بررسي قرار مي‌گيرند. ساختار Point داراي سازنده ايست كه مقادير داده شده با آنرا به فيلدهاي x و y تخصيص مي‌دهد. اين ساختار همچنين داراي متد Add() مي‌باشد كه ساختار Point ديگري را دريافت مي‌كند و آنرا به struct كنوني مي‌افزايد و سپس struct جديدي را باز مي‌گرداند.

توجه نماييد كه ساختار Point جديدي درون متد Add() تعريف شده است. توجه كنيد كه در اينجا همانند كلاس، نيازي به استفاده از كلمه كليدي new جهن ايجاد يك شيء جديد نمي‌باشد. پس از آنكه نمونة جديدي از يك ساختار ايجاد

براي آن در نظر گرفته مي‌شود. سازندة بدون پارامتر كليه مقادير فيلدهاي ساختار را به مقادير پيش فرض تغيير مي‌دهد. بعنوان مثال فيلدهاي صحيح به صفر و فيلدهاي Boolean به false تغيير مي‌كنند. تعريف سازندة بدون پارامتر براي يك ساختار صحيح نمي‌باشد. (يعني شما نمي‌توانيد سازندة بدون پارامتر براي يك struct تعريف كنيد.)

ساختارها (structs) با استفاده از عملگر new نيز قابل نمونه‌گيري هستند (هر چند نيازي به استفاده از اين عملگر نيست.) در مثال ۱-۱۱ pt1 و pt2 كه ساختارهايي از نوع Point هستند، با استفاده از سازندة موجود درون ساختار Point مقداردهي مي‌شوند. سومين ساختار از نوع Point، pt3 است و از سازندة بدون پارامتر استفاده مي‌كند زيرا در اينجا مقدار آن اهميتي ندارد. سپس متد Add() از ساختار pt1 فراخوانده مي‌شود و ساختار pt2 را بعنوان پارامتر دريافت مي‌كند. نتيجه به pt3 تخصيص داده مي‌شود، اين امر نشان مي‌دهد كه يك ساختار مي‌تواند همانند ساير انواع مقداري مورد استفاده قرار گيرد. خروجي مثال ۱-۱۱ در زير نشان داده شده است:

pt3: 3: 3

يكي ديگر از تفاوتهاي ساختار و كلاس در اينست كه ساختارها نمي‌توانند داراي تخريب كننده (deconstructor) باشند. همچنين ارثبري در مورد ساختارها معني ندارد. البته امكان ارثبري بين ساختارها و interface ها وجود دارد. يك interface نوع مرجعي زبان C# است كه داراي اعضايي بدون پيادەسازي است. هر كلاس و يا ساختاري كه از يك interface ارثبري نمايد بايد تمامي متدهاي آنرا پيادە سازي كند. دربارة interface ها در آينده صحبت خواهيم كرد.

نكات مهم و مطالب كمكي

۱. تفاوتهاي اصلي بين كلاس و ساختار در چيست؟

همانطور كه بطور مختصر در بالا نيز اشاره شد، از نظر نوشتاري (syntax) struct و كلاس بسيار شبيه به يكديگر هستند اما داراي تفاوتهاي بسيار مهمي با يكديگر مي‌باشند.
همانطور كه قبلاً نيز اشاره شد شما نمي‌توانيد براي يك struct سازنده‌اي تعريف كنيد كه بدون پارامتر است، يعني براي ايجاد سازنده براي يك struct حتماً بايد اين سازنده داراي پارامتر باشد. به قطعه كد زير توجه كنيد:

```
struct Time
{
 public Time() {. .. } // خطاي زمان كامپايل رخ مي‌دهد
```

}

پس از اجراي كد فوق كامپايلر خطايي را ايجاد خواهد كرد بدين عنوان كه سازندة struct حتماً بايد داراي پارامتر باشد. حال اگر بجاي struct از كلمه كليدي calss استفاده كرده بوديم اين كد خطايي را ايجاد نمي‌كرد. در حقيقت تفاوت در اينست كه در مورد struct، كامپايلر اجازة ايجاد سازندة پيش فرض جديدي را به شما نمي‌دهد ولي در مورد كلاس چنين نيست. هنگام اعلان كلاس در صورتيكه شما سازندة پيش فرضي اعلان نكرده باشيد، كامپايلر سازنده‌اي پيش فرض براي آن در نظر مي‌گيرد ولي در مورد struct تنها سازندة پيش فرضي معتبر است كه كامپايلر آنرا ايجاد نمايد نه شما !

يكي ديگر از تفاوتهاي بين كلاس و struct در آن است كه، اگر در كلاس برخي از فيلدهاي موجود در سازندة كلاس را مقداردهي نكنيد، كامپايلر مقدار پيش فرض صفر، false و يا null را براي آن فيلد در نظر خواهد گرفت ولي در struct تمامي فيلدهاي سازنده بايد بطور صريح مقداردهي شوند و درصورتيكه شما فيلدي را مقداردهي نكنيد كامپايلر هيچ مقداري را براي آن در نظر نخواهدگرفت و خطاي زمان كامپايل رخ خواهد داد. بعنوان مثال در كد زير اگر Time بصورت كلاس تعريف شده بود خطايي رخ نمي‌داد ولي چون بصورت struct تعريف شده خطاي زمان كامپايل رخ خواهد داد:

```
struct Time
{
 public Time(int hh, int mm)
 {
  hours = hh;
  minutes = mm;
 } // خطاي زمان كامپايلي بدين صورت رخ مي‌دهد: seconds not initialized
 :
 private int hours, minutes, seconds;
}
```

تفاوت ديگر كلاس و struct در اينست كه در كلاس مي‌توانيد در هنگام اعلان فيلدها را مقداردهي كنيد حال آنكه در struct چنين عملي باعث ايجاد خطاي زمان كامپايل خواهد شد. همانند كدهاي فوق، در كد زير اگر از كلاس بجاي struct استفاده شده بود خطا رخ نمي‌داد:

```
struct Time
{
 :
 private int hours = 0; // كامپايل رخ مي‌دهد خطاي زمان
 private int minutes;
```

```
}
```

آخرين تفاوت بين كلاس و struct كه ما به آن خواهيم پرداخت در مورد ارثبري است. كلاسها ميتوانند از كلاس پاية خود ارثبري داشته باشند در حاليكه ارثبري در struct ها معنايي ندارد و يك struct تنها ميتواند از واسطها (interface) ارثبري داشته باشد.

۲. پس از ايجاد يك ساختار چگونه ميتوان از آن استفاده نمود؟

همانطور كه گفتيم، ساختارها روشي براي ايجاد انواع جديد مقدار (Value Types) هستند. از اينرو پس از ايجاد يك ساختار ميتوان از آن همانند ساير انواع مقداري استفاده نمود. براي استفاده از يك ساختار ايجاد شده كافيست تا نام آنرا قبل از متغير مورد نظر قرار دهيم تا متغير مورد نظر از نوع آن ساختار خاص تعريف شود.

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

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

آخرين نكتهاي كه در مورد ساختارها براي چندمين بار اشاره ميكنم انست كه، ساختارها انواع مقداري هستند و مستقيماً مقدار را در خود نگه ميدارند و از اينرو در stack نگهداري ميشوند. استفاده از ساختارها همانند ساير انواع مقداري است.

فصل دوازدهم – واسطها (Interfaces)

در اين درس با واسطها در زبان C# آشنا خواهيم شد. اهداف اين درس بشرح زير مي‌باشند:

۱- آشنايي با مفهوم كلي واسطها
۲- تعريف يك واسط
۳- استفاده از يك interface
٤- پياده‌سازي ارثبري در interface ها
٥- نكات مهم و پيشرفته
٦- مثالي كاربردي از واسطها
٧- منابع مورد استفاده

واسطها از لحاظ ظاهري بسيار شبيه به كلاس هستند با اين تفاوت كه داراي هيچ گونه پياده‌سازي نمي‌باشند. تنها چيزي كه در interface به چشم مي‌خورد تعاريفي نظير رخدادها، متدها، انديكسرها و يا property ها است. يكي از دلايل اينكه واسطها تنها داراي تعاريف هستند و پياده سازي ندارند آنست كه يك interface مي‌توان توسط چندين كلاس يا property مورد ارثبري قرار گيرد، از اينرو هر كلاس يا property خواستار آنست كه خود به پياده‌سازي اعضا بپردازد.

حال بايد ديد چرا با توجه به اينكه interface ها داراي پياده سازي نيستند مورد استفاده قرار مي‌گيرند يا بهتر بگوئيم سودمندي استفاده از interface ها در چيست؟ تصور كنيد كه در يك برنامه با مولفه‌هايي سروكار داريد كه متغيرند ولي داراي فيلدها يا متدهايي با نامهاي يكساني هستند و بايد نام اين متدها نيز يكسان باشد. با استفاده از يك interface مناسب مي‌توان تنها متدها و يا فيلدهاي مورد نظر را اعلان نمود و سپس كلاسها و يا property هاي مورد از آن interface ارثبري نمايند. در اين حالت تمامي كلاسها و property ها داراي فيلدها و يا متدهايي همنام هستند ولي هر يك پياده‌سازي خاصي از آنها را اعمال مي‌نمايند.

نكته مهم ديگر درباره استفاده و كاربرد interface ها، استفاده و كاربرد آنها در برنامه‌هاي بزرگي است كه برنامه‌ها و يا اشياؤ مختلفي در تماس و تراكنش (transact) هستند. تصور كنيد كلاسي در يك برنامه با كلاسي ديگر در برنامه‌اي ديگر در ارتباط باشد. فرض كنيد اين كلاس متدي دارد كه مقداري از نوع int بازميگرداند. پس از مدتي طراح برنامه به اين نتيجه مي‌رسد كه استفاده از int پاسخگوي مشكلش نيست و بايد از long استفاده نمايد. حال شرايط را در نظر بگيريد كه براي تغيير يك چنين مسئله ساده‌اي چه مشكل بزرگي پيش خواهد آمد. تمامي فيلدهاي مورتبط با اين متد بايد تغيير داده شوند. در ضمن از مسئله side effect نيز نمي‌توان چشم پوشي

پیش بيني نشده كه متغير يا فيلدي بر روي متغير يا فيلدي ديگر اعمال مي‌كند، در اصطلاح side effect گفته مي‌شود.) حال فرض كنيد كه در ابتدا interface اي طراحي شده بود. درصورت اعمال جزئيترين تغيير در برنامه مشكل تبديل int به long قابل حل بود، چراكه كاربر يا برنامه و در كل user برنامه در هنگام استفاده از يك interface با پيادهسازي پشت پرده آن كاري ندارد و يا بهتر بگوئيم امكان دسترسي به آن را ندارد. از اينرو اعمال تغييرات درون آن تاثيري بر رفتار كاربر نخواهد داشت و حتي كاربر از آن مطلع نيز نمي‌شود. در مفاهيم كلي شيء گرايي، interface ها يكي از مهمترين و كاربردي ترين اجزاء هستند كه در صورت درك صحيح بسيار مفيد واقع مي‌شوند. يكي از مثالهاي مشهود درباره interface ها (البته در سطحي پيشرفته تر و بالاتر) رابطهاي كاربر گرافيكي (GUI) هستند. كاربر تنها با اين رابط سروكار دارد و كاري به نحوه عمليات پشت پرده آن ندارد و اعمال تغييرات در پيادهسازي interface كاربر را تحت تاثير قرار نمي‌دهد.

از ديدگاه تكنيكي، واسطها بسط مفهومي هستند كه از آن به عنوان انتزاع (Abstract) ياد مي‌كنيم. در كلاسهاي انتزاعي (كه با كلمه كليد abstract مشخص مي‌شدند.) سازندة كلاس قدر بود تا فرم كلاس خود را مشخص نمايد: نام متدها، نوع بازگشتي آنها و تعداد و نوع پارامتر آنها، اما بدون پيادهسازي بدنه متد. يك interface همچنين مي‌تواند داراي فيلدهايي باشد كه تمامي آنها static و final هستند. يك interface تنها يك فرم كلي را بدون پيادهسازي به نمايش مي‌گذارد.

از اين ديدگاه، يك واسط بيان مي‌دارد كه: " اين فرم كلي است كه تمامي كلاسهايي كه اين واسط را پيادهسازي مي‌كنند، بايد آنرا داشته باشند." از سوي ديگر كلاسها و اشياء ديگري كه از كلاسي كه از يك واسط مشتق شده استفاده مي‌كنند، مي‌دانند كه اين كلاس حتماً تمامي متدها و اعضاي واسط را پيادهسازي مي‌كند و مي‌توانند به راحتي از آن متدها و اعضا استفاده نمايند. پس به طور كلي مي‌توانيم بگوئيم كه واسطها **بمنظور ايجاد يك پروتكل (protocol) بين كلاسها مورد استفاده قرار مي‌گيرند.** (همچنان كه برخي از زبانهاي برنامه‌سازي بجاي استفاده از كلمه كليدي interface از protocol استفاده مي‌نمايند.)

به دليل اينكه كلاسها و ساختارهايي كه از interface ها ارثبري مي‌كنند موظف به پيادهسازي و تعريف آنها هستند، قانون و قاعده‌اي در اين باره ايجاد مي‌گردد. براي مثال اگر كلاس A از واسط IDisposable ارثبري كند، اين ضمانت

عضو interface نيز مي‌باشد. هر كدي كه مي‌خواهد از كلاس A استفاده كند، ابتدا چك مي‌نمايد كه آيا كلاس A واسط IDisposable را پياده‌سازي نموده يا خير. اگر پاسخ مثبت باشد آنگاه كد متوجه مي‌شود كه مي‌تواند از متد A.Dispose() نيز استفاده نمايد. در زير نحوه اعلان يك واسط نمايش داده شده است.

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

در اين مثال نحوه اعلان واسطي با نام IMyInterface نشان داده شده است. يك قاعده (نه قانون!) براي نامگذاري واسطها آنست كه نام واسطها را با "I" آغاز كنيم كه اختصار كلمه interface است. در interface اين مثال تنها يك متد وجود دارد. اين متد مي‌توان هر متدي با انواع مختلف پارامترها و نوع بازگشتي باشد. توجه نماييد همانطور كه گفته شد اين متد داراي پياده سازي نيست و تنها اعلان شده است. نكته ديگر كه بايد به ان توجه كنيد آنست كه اين متد به جاي داشتن {} به عنوان بلوك خود، داراي ; در انتهاي اعلان خود مي‌باشد. علت اين امر آنست كه interface تنها نوع بازگشتي و پارامترهاي متد را مشخص مي‌نمايد و كلاس يا شي‌اي كه از آن ارث مي برد بايد آنرا پياده سازي نمايد. مثال زير نحوه استفاده از اين واسط را نشان مي‌دهد.

مثال ۱-۱۲: استفاده از واسطها و ارثبري از آنها

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

در اين مثال، كلاس InterfaceImplementer همانند ارثبري از يك كلاس، از واسط IMyInterface ارثبري كرده است. حال كه اين كلاس از واسط مورد نظر ارثبري كرده است، بايد، توجه نماييد بايد، تمامي اعضاي آنرا پياده‌سازي كند. در اين مثال اين عمل با پياده‌سازي تنها عضو واسط يعني متد MethodToImplement() انجام گرفته است. توجه نماييد كه

و نوع پارامترها شبیه به اعلان موجود در واسط باشد، کوچکترین تغییري باعث ایجاد خطاي کامپایلر مي‌شود. مثال زیر نحوه ارثبري واسطها از یکدیگر نیز نمایش داده شده است.

مثال ۲-۱۲: ارثبري 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 ۲-۱۲ داراي ۲ واسط است: یکي IMyInterface و واسطي که از آن ارث مي‌برد یعني IParentInterface. هنگامیکه واسطي از واسط دیگري ارثبري مي‌کند، کلاس یا ساختاري که این واسطها را پیاده‌سازي مي‌کند، باید تمامي اعضاي واسطهاي موجود در سلسله مراتب ارثبري را پیاده‌سازي نماید. در مثال ۲-۱۲، چون کلاس InterfaceImplementer از واسط IMyInterface ارثبري نموده، پس از واسط IParentInterface نیز ارثبري دارد، از اینرو باید کلیه اعضاي این دو واسط را پیاده‌سازي نماید.

۱- با استفاده از کلمه کلید interface در حقیقت یک نوع مرجعي (Reference Type) جدید ایجاد نموده اید.

۲- از لحاظ نوع ارتباطي که واسطها و کلاسها در ارثبري ایجاد مي‌نمایند باید به این نکته اشاره کرد که، ارثبري از کلاس رابطه "است" یا "بودن" (is-a relation) را ایجاد مي‌کند (ماشین یک وسیله نقلیه است) ولي ارثبري از یک واسط یا interface نوع خاصي از رابطه، تحت عنوان "پیاده‌سازي" (implement relation) را ایجاد مي‌کنند. ("مي‌توان ماشین را با وام بلند مدت خرید" که در این جمله ما شین مي‌تواند خریداري شدن بو سیله وام را پیاده‌سازي کند.)

۳- فرم کلي اعلان interface ها بشکل زیر است:

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

که در اعضاي آن بشرح زیر مي باشند:

attributes: صفتهاي واسط

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

interface-name: نام واسط

:base-list: لیست واسطهایي که این واسط آنها را بسط مي‌دهد.

Interface-body: بدنه واسط که در آن اعضاي آن مشخص مي‌شوند

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

٤- هدف از ایجاد یک interface تعیین توانائیهاییست که مي‌خواهیم در یک کلاس وجود داشته باشند.

٥- به مثالي در زمینه استفاده از واسطها توجه کنید:

فرض کنید مي‌خواهید واسطي ایجاد نمایید که متدها و property هاي لازم براي کلاسي را که مي‌خواهد قابلیت خواندن و نوشتن از/به یک پایگاه داده یا هر فایلي را داشته باشد، توصیف نماید. براي این منظور مي‌توانید از واسط IStorable استفاده نمایید.

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

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

حال مي‌خواهید کلاسي با عنوان Document ایجاد نمایید که این کلاس باید قابلیت خواندن و نوشتن از/ به پایگاه داده را داشته باشد، پس مي‌توانید کلاس را از روي واسط IStorable پیاده‌سازي کنید.

```
public class Document: IStorable
{
```

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

حال بعنوان طراح برنامه، شما وظيفه داري تا به پيادهسازي اين واسط بپردازيد، بطوريكه كليه نيازهاي شما را برآورده نمايد. نمونهاي از اين پيادهسازي در مثال ۳-۱۲ آورده شده است.

مثال ۳-۱۲: پيادهسازي واسط و ارثبري - مثال كاربردي

```
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);
  }
}
```

خروجي برنامه نيز بشكل زير 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
```

٦- در مثال فوق توجه نماييد كه براي متدها واسط IStorable هيچ سطح دسترسي (public,private و. . .) در نظر گرفته نشده است. در حقيقت تعيين سطح دسترسي باعث ايجاد خطا مي‌شود چراكه هدف اصلي از ايجاد يك واسط ايجاد شيء است كه تمامي اعضاي آن براي تمامي كلاسها قابل دسترسي باشند.

٧- توجه نماييد كه از روي يك واسط نمي‌توان نمونه‌اي جديد ايجاد كرد بلكه بايد كلاسي از آن ارثبري نمايد.

٨- كلاسي كه از واسط ارثبري مي‌كند بايد تمامي متدهاي آنرا دقيقا همان گونه كه در واسط مشخص شده پياده‌سازي نمايد. به بيان كلي، كلاسي كه از يك واسط ارث مي‌برد، فرم و ساختار كلي خود را از واسط مي‌گيرد و نحوه رفتار و پياده‌سازي آنرا خود انجام مي‌دهد.

نکته مهم قبل از مطالعه این درس

توجه نمایید، delegate ها و رخدادها بسیار با یکدیگر در تعاملاند، از اینرو در برخی موارد، قبل از آموزش و بررسی رخدادها، به ناچار، از آنها نیز استفاده شده و یا به آنها رجوع شده است. رخدادها در قسمت انتهایی این درس مورد بررسی قرار میگیرند، 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 اینو مفهوم با یکدیگر آورده شده اند ولی درک رخدادها مستلزم درک و فراگیری کامل delegate هاست، از اینرو مطالب مربوط به delegate ها را در ابتدا قرار داده ام.

هدف ما در این درس به شرح زیر است:

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

طی درسهای گذشته، چگونگی ایجاد و پیادسازی انواع مرجعی (Reference Type) را با استفاده از ساختارهای زبان C#، یعنی کلاسها (Class) و واسطها (Interface)، فرا گرفتید. همچنین فرا گرفتید که با استفاده از این انواع مرجعی، میتوانید نمونه‌های جدیدی از اشیاء را ایجاد کرده و نیازهای توسعه 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.

تعريف مي‌كرديم تا فرم كلي داشته باشيم و تمام اشياء خود به پياده‌سازي اين صفا و رفتارها مي‌پرداختند. در اين درس با يكي ديگر از انواع مرجعي (Reference Type) در زبان C# آشنا خواهيد شد.

مقدمه‌اي بر رخدادها و delegate ها

در گذشته، پس از اجراي يك برنامه، برنامه مراحل اجراي خود را مرحله به مرحله اجرا مي‌نمود تا به پايان برسد. در صورتيكه نياز به ارتباط و تراكنش با كاربر نيز وجود داشت، اين امر محدود و بسيار كنترل شده صورت مي‌گرفت و معمولاً ارتباط كاربر با برنامه تنها پر كردن و يا وارد كردن اطلاعات خاصي در فيلدهايي مشخص بود.

امروزه با پيشرفت كامپيوتر و گسترش تكنولوژيهاي برنامه نويسي و با ظهور رابطهاي كاربر گرافيكي (GUI) ارتباط بين كاربر و برنامه بسيار گسترش يافته و ديگر اين ارتباط محدود به پر كردن يكسري فيلد نيست، بلكه انواع عمليات از سوي كاربر قابل انجام است. انتخاب گزينه‌اي خاص در يك منو، كليك كردن بر روي دكمه‌ها براي انجام عملياتي خاص و.... رهيافتي كه امروزه در برنامه‌نويسي مورد استفاده است، تحت عنوان "برنامه‌نويسي بر پايه رخدادها" (Event-Based Programming) شناخته مي‌شود. در اين رهيافت برنامه همواره منتظر انجام عملي از سوي كاربر مي‌ماند و پس از انجام عملي خاص، رخداد مربوط به آن را اجرا مي‌نمايد. هر عمل كاربر باعث اجراي رخدادي مي‌شود. در اين ميان برخي از رخدادها بدون انجام عملي خاص از سوي كاربر اجرا مي‌شوند، همانند رخدادهاي مربوط به ساعت سيستم كه مرتباً در حال اجرا هستند.

رخدادها (Events) بيان اين مفهوم هستند كه در صورت اتفاق افتادن عملي در برنامه، كاري بايد صورت گيرد. در زبان C# مفاهيم Event و Delegate دو مفهوم بسيار وابسته به يكديگر هستند و با يكديگر در تعامل مي‌باشند. براي مثال، مواجهه با رخدادها و انجام عمل مورد نظر در هنگام اتفاق افتادن يك رخداد، نياز به يك event handler دارد تا در زمان بروز رخداد، بتوان به آن مراجعه نمود. Event handler ها در C# معمولاً با delegate ها ساخته مي‌شوند.

از delegate، مي‌توان به عنوان يك Callback ياد نمود، بدين معنا كه يك كلاس مي‌تواند به كلاسي ديگر بگويد: "اين عمل خاص را انجام بده و هنگاميكه عمليات را انجام دادي منرا نيز مطلع كن". با استفاده از delegate ها، همچنين مي‌توان

باشند.

Delegate

Delegate ها، يكي ديگر از انواع مرجعي زبان C# هستند كه با استفاده از آنها مي‌توانيد مرجعي به يك متد داشته باشيد، بدين معنا كه delegate ها، آدرس متدي خاص را در خود نگه ميدارند. در صورتيكه قبلاً با زبان C برنامه‌نويسي كرده‌ايد، حتماً با اين مفهوم آشنايي داريد. در زبان C اين مفهوم با اشاره‌گرها (pointer) بيان مي‌شود. اما براي افرادي كه با زبانهاي ديگري برنامه‌نويسي مي‌كرده‌اند و با اين مفهوم مانوس نيستند، شايد اين سوال مطرح شود كه چه نيازي به داشتن آدرس يك متد وجود دارد. براي پاسخ به اين سوال اندكي بايد تامل نماييد.

بطور كلي مي‌توان گفت كه delegate نوعي است شبيه به متد و همانند آن نيز رفتار مي‌كند. در حقيقت delegate انتزاعي (Abstraction) از يك متد است. در برنامه‌نويسي ممكن به شرايطي برخورد كرده باشيد كه در آنها مي‌خواهيد عمل خاصي را انجام دهيد اما دقيقاً نمي‌دانيد كه بايد چه متد يا شيء‌اي را براي انجام آن عمل خاص مورد استفاده قرار دهيد. در برنامه‌هاي تحت ويندوز اين گونه مسائل مشهودتر هستند. براي مثال تصور كنيد در برنامه شما، دكمه‌اي قرار دارد كه پس از فشار دادن اين دكمه توسط كاربر شيء‌اي يا متدي بايد فراخواني شود تا عمل مورد نظر شما بر روي آن انجام گيرد. مي‌توان بجاي اتصال اين دكمه به شيء يا متد خاص، آنرا به يك delegate مرتبط نمود و سپس آن delegate را به متد يا شيء خاصي در هنگام اجراي برنامه متصل نمود.

ابتدا، به نحوه استفاده از متدها توجه نماييد. معمولاً، براي حل مسايل خود الگوريتم هايي طراحي مي‌نائيم كه اين الگوريتمهاي كارهاي خاصي را با استفاده از متدها انجام مي‌دهد، ابتدا متغيرهايي مقدار دهي شده و سپس متدي جهت پردازش آنها فراخواني مي‌گردد. حال در نظر بگيريد كه به الگوريتمي نياز داريد كه بسيار قابل انعطاف و قابل استفاده مجدد (reusable) باشد و همچنين در شرايط مختلف قابليت هاي مورد نظر را در اختيار شما قرار دهد. تصور كنيد، به الگوريتمي نياز داريد كه از نوعي از ساختمان داده پشتيباني كند و همچنين مي‌خواهيد اين ساختمان داده را در مواردي مرتب (sort) نماييد، بعلاوه ميخواهيد تا اين

انواع موجود در اين ساختمان داده را نداند، چگونه مي‌خواهيد الگوريتمي جهت مقايسه عناصر آن طراحي كنيد؟ شايد از يك حلقه if/then/else و يا دستور switch براي اين منظور استفاده كنيد، اما استفاده از چنين الگوريتمي محدوديتي براي ما ايجاد خواهد كرد. روش ديگر، استفاده از يك واسط است كه داراي متدي عمومي باشد تا الگوريتم شما بتواند آنرا فراخواني نمايد، اين روش نيز مناسب است، اما چون مبحث ما در اين درس delegate ها هستند، مي‌خواهيم مسئله را از ديدگاه delegate ها مورد بررسي قرار دهيم. روش حل مسئله با استفاده از آنها اندكي متفاوت است.

روش ديگر حل مسئله آنست كه، مي‌توان delegate ي را به الگوريتم مورد نظر ارسال نمود و اجازه داد تا متد موجود در آن،عمل مورد نظر ما را انجام دهد. چنين عملي در مثال ۱-۱۴ نشان داده شده است.
(به صورت مسئله توجه نماييد: ميخواهيم مجموعه‌اي از اشياء را كه در يك ساختمان داده قرار گرفته‌اند را مرتب نمائيم. براي اينكار نياز به مقايسه اين اشياء با يكديگر داريم. از آنجائيكه اين اشياء از انواع (type) مختلف هستند به الگوريتمي نياز داريم تا بتواند مقايسه بين اشياء نظير را انجام دهد. با استفاده از روشهاي معمول اين كار امكان پذير نيست، چراكه نمي‌توان اشيائي از انواع مختلف را با يكديگر مقايسه كرد. براي مثال شما نمي‌توانيد نوع عددي int را با نوع رشته‌اي string مقايسه نماييد. به همين دليل با استفاده از delegate ها به حل مسئله پرداخته‌ايم. به مثال زير به دقت توجه نماييد تا بتوانيد به درستي مفهوم delegate را درك كنيد.)
مثال ۱-۱۳: اعلان و پياده‌سازي يك delegate

```
using System;

// delegate در اينجا اعلان مي‌گردد.
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 n2 = ((Name)name2).FirstName;
    if (String.Compare(n1, n2) > 0)
    {
      return 1;
    }
    else if (String.Compare(n1, n2) < 0)
    {
      return -1;
    }
    else
    {
      return 0;
    }
  }
  public override string ToString()
  {
    return FirstName + " " + LastName;
  }
}

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

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

  public void Sort(Comparer compare)
  {
    object temp;
    for (int i=0; i < names.Length; i++)
```

```
      for (int j=i; j < names.Length; j++)
      {
        //از compare همانند یك متد استفاده مي‌شود
        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());
    }
  }
}
```

اولين اعلان در اين برنامه، اعلان delegate است. اعلان delegate بسيار شبيه به اعلان متد است، با اين تفاوت كه داراي كلمه كليدي delegate در اعلان است و در انتهاي اعلان آن ";" قرار مي‌گيرد و نيز پياده‌سازي ندارد. در زير اعلان delegate كه در مثال ۱-۱۳ آورده شده را مشاهده مي‌نماييد:

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

اين اعلان، مدل متدي را كه delegate مي‌تواند به آن اشاره كند را تعريف مي‌نمايد. متدي كه مي‌توان از آن بعنوان delegate handler براي Comparer استفاده نمود، هر متدي مي‌تواند باشد اما حتماً بايد پارامتر اول و دوم آن از نوع object بوده و مقداري از نوع int بازگردانند. در زير متدي كه بعنوان delegate handler در مثال ۱-۱۳ مورد استفاده قرار گرفته است، نشان داده شده است:

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

براي استفاده از delegate مي‌بايست نمونه‌اي از آن ايجاد كنيد. ايجاد نمونه جديد از delegate همانند ايجاد نمونه‌اي جديد از يك كلاس است كه به همراه پارامتري جهت تعيين متد delegate handler ايجاد مي‌شود:

در مثال ۱-۱۳، cmp بعنوان پارامتري براي متد Sort() مورد استفاده قرار گرفته است. به روش ارسال delegate به متد Sort() توجه نماييد:

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

با استفاده از اين تكنيك، هر متد handler به سادگي در زمان اجرا به متد Sort() قابل ارسال است. براي مثال مي‌توان handler ديگري با نام CompareLastNames() تعريف كنيد، نمونه جديدي از Comparer را با اين پارامتر ايجاد كرده و سپس آنرا به متد Sort() ارسال نماييد.

درك سودمندي delegate ها

براي درك بهتر delegate ها به بررسي يك مثال مي‌پردازيم. در اينجا اين مثال را يكبار بدون استفاده از delegate و بار ديگر با استفاده از آن حل كرده و بررسي مي‌نمائيم. مطالب گفته شده در بالا نيز به نحوي مرور خواهند شد. توجه نماييد، همانطور كه گفته شد delegate ها و رخدادها بسيار با يكديگر در تعامل‌اند، از اينرو در برخي موارد به ناچار از رخدادها نيز استفاده شده است. رخدادها در قسمت انتهايي اين درس آورده شده‌اند، از اينرو در صورتيكه در برخي موارد دچار مشكل شديد و يا درك مطلب برايتان دشوار بود، ابتدا كل درس را تا انتها مطالعه نماييد و سپس در بار دوم با ديدي جديد به مطالب و مفاهيم موجود در آن نگاه كنيد. در اغلب كتابهاي آموزشي زبان C# نيز ايندو مفهوم با يكديگر آورده شده‌اند ولي درك رخدادها مستلزم درك و فراگيري كامل delegate هاست، از اينرو مطالب مربوط به delegate ها را در ابتدا قرار داده‌ام.

حل مسئله بدون استفاده از delegate

فرض كنيد، ميخواهيد برنامه بنويسيد كه عمل خاصي را هر يك ثانيه يكبار انجام دهد. يك روش براي انجام چنين عملي آنست كه، كار مورد نظر را در يك متد پياده‌سازي نماييد و سپس با استفاده از كلاسي ديگر، اين متد را هر يك ثانيه يكبار فراخواني نمائيم. به مثال زير توجه كنيد:

```
class Ticker
{
 :
 public void Attach(Subscriber newSubscriber)
 {
  subscribers.Add(newSubscriber);
 }
 public void Detach(Subscriber exSubscriber)
```

```
 subscribers.Remove(exSubscriber);
 }
 // هر ثانیه فراخواني ميگردد Notify
 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 نمونه هاي جدید کلاس Subscriber ميتوانند از کلاس Ticker استفاده نمایند. اگر در برنامه کلاس دیگري داشته باشید که بخواهید آن کلاس نیز هر یك ثانیه یکبار اجرا شود، ميبایست کلاس جدیدي شبیه به Ticker ایجاد کنید. براي بهینه کردن این مسئله ميتوانید از یك واسط (Interface) نیز کمك بگیرید. براي این منظور ميتوان متد Tick را درون واسطي قرار داد و سپس کلاس Ticker را به این واسط مرتبط نمود.

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

```
{
 public void Attach(Tickable newSubscriber)
 {
  subscribers.Add(newSubscriber);
 }
 public void Detach(Tickable exSubscriber)
 {
  subscribers.Remove(exSubscriber);
 }
 // هر ثانیه فراخوانی میگردد Notify
 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.

استفاده از واسطها در برنامه‌ها، مطمئناً روشي بسيار خوب است، اما كامل نبوده اشكالاتي دارد. مشكل اول آنست كه اين روش بسيار كلي و عمومي است. تصور نماييد مي‌خواهيد از تعداد زيادي از سرويسها استفاده نماييد(بعنوان مثال در برنامه‌هاي مبتني بر GUI. در اينگونه برنامه‌ها هجم عظيمي از رخدادها وجود دارند كه مي‌بايست با تمامي آنها در ارتباط باشيد.) مشكل ديگر آنست كه استفاده از واسط، بدين معناست كه متد Tick بايد متدي public باشد، از اينرو هر كدي مي‌تواند Clock.Tick را در هر زماني فراخواني نمايد. روش مناسب تر آنست كه مطمئن شويم تنها اعضايي خاص قادر به فراخواني و دسترسي به Clock.Tick هستند. با استفاده از delegate تمامي اين امكانات براي ما فراهم خواهد شد و برنامه‌هايي با ايمني بالاتر و پايدارتر مي‌توانيم داشته باشيم.

اعلان Delegate

در مثال ما، متد Tick از واسط Tickable از نوع void بود و هيچ پارامتري دريافت نمي‌كرد:

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

براي اين متد مي‌توان delegate ي تعريف نمود كه ويژگيهاي آنرا داشته باشد:

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

همانطور كه قبلاً نيز گفته شد، اين عمل نوع جديدي را ايجاد مي‌نمايد كه مي‌توان از آن همانند ساير انواع استفاده نمود. مثلاً مي‌توان آنرا بعنوان پارامتري براي يك متد در نظر گرفت:

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

فراخواني delegate

قدرت و توانايي delegate زماني مشهود مي‌گردد كه مي‌خواهيد از آن استفاده نماييد. براي مثال، با متغير param در مثال قبل چكار مي‌توانيد انجام دهيد؟ اگر param متغيري از نوع int بود، از مقدار آن استفاده مي‌كرديد و با استفاده از عملگرهايي نظير +، - و يا عملگرهاي مقايسه‌اي، عملي خاص را بر روي آن انجام مي‌داديد. اما حال كه param

است و همانطور كه گفته شد، delegate انتزاعي از يك متد است، پس هر عملي كه متد انجام مي‌دهد، delegate نيز مي‌تواند انجام دهد. با استفاده از پرانتز، مي‌توان از delegate استفاده نمود:

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

نكته: همانطور كه اشاره شد، delegate يكي از انواع مرجعي است از اينرو مقدار آن مي‌تواند برابر با Null باشد. در مثال فوق، اگر مقدار param برابر با Null باشد، كامپايلر خطاي NullReferenceException را ايجاد مي‌نمايد.

همانند متدها، delegate ها بايد بطور كامل و صحيح فراخواني گردند. با توجه به اعلان Tick، در زمان فراخواني اين delegate، مثلاً param، بايد توجه داشت كه هيچ پارامتري را نمي‌توان به آن ارسال نمود و نمي‌توان آنرا به متغيري نسبت داد چراكه اين delegate بصورت void اعلان شده و مقدار بازگشتي ندارد.

```
void Example(Tick param)
{
 param(42);    // خطاي زمان كامپايل رخ مي‌دهد
 int hhg = param();  // خطاي زمان كامپايل رخ مي‌دهد
 Console.WriteLine(param()); // خطاي زمان كامپايل رخ مي‌دهد
}
```

توجه نماييد كه delegate را به هر نحوي مي‌توانيد اعلان نماييد. براي مثال به نسخة ديگري از Tick توجه كنيد:

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

اما به ياد داشته باشيد كه همانند متد، در هنگام استفاده از آن بايد پارامترهاي صحيح به آن ارسال نماييد:

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

با استفاده از delegate مي‌توانيد كلاس Ticker را پياده‌سازي كنيد:

```
delegate void Tick(int hours, int minutes, int seconds);
class Ticker
{
 ⋮
 public void Attach(Tick 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 نمونه‌هاي جديد از يك delegate

آخرين كاري كه بايد انجام دهيد، ايجاد نمونه‌هاي جديد از delegate ساخته شده است. يك نمونة جديد از يك delegate، تنها انتزاعي از يك متد است كه با نامگذاري آن متد ايجاد مي‌شود.

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

با توجه به ساختار Tick، ملاحظه مي‌نماييد كه متد RefreshTime كاملاً با اين delegate همخواني دارد:

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

و اين بدين معناست كه مي‌توان نمونة جديد از Tick ايجاد كرد كه انتزاعي از فراخواني RefreshTime در شيء خاصي از Clock است.

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

حال كه m، ايجاد شد، مي‌توانيد از آن بصورت زير استفاده نماييد:

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

اين دستور در حقيقت كار دستور زير را انجام مي‌دهد (چون m دقيقاً انتزاع آن است):

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

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.

رخداد خاصي، فعال مي‌شود و عملي را انجام مي‌دهد. معمولاً براي فعال شده event از دو عبارت fires و raised استفاده مي‌شود. هر متدي كه بخواهد، ميتواند در ليست رخداد ثبت شده و به محض اتفاق افتادن آن رخداد، از آن مطلع گردد.

بطور كلي مي‌توان گفت كه يك رخداد همانند يك فيلد اعلان مي‌شود با اين تفاوت مهم كه نوع آنها حتماً بايد يك delegate باشد.

Delegate و رخدادها در كنار يكديگر كار مي‌كنند تا قابليت‌هاي يك برنامه را افزايش دهند. اين پروسه با شروع يك كلاس كه يك رخداد را تعريف مي‌كند، آغاز مي‌شود. هر كلاسي، كه اين رخداد را درون خود داشته باشد، در آن رخداد ثبت شده است و مي‌تواند متدي را به آن رخداد تخصيص دهد. اين عمل با استفاده از delegate ها صورت مي‌پذيرد، بدين معني كه delegate متدي را كه براي رخداد ثبت مي‌شود را تعيين مي‌نمايد. Delegate ها مي‌توانند هر يك از delegate هاي از پيش تعريف شدةNET. و يا هر delegate ي باشند كه توسط كاربر تعريف شده است. بطور كلي، delegate ي را به رخدادي تخصيص مي‌دهيم تا متدي را كه بهنگام روي دادن رخداد فراخواني مي‌شود، معين گردد. مثال زير روش تعريف رخداد را نشان مي‌دهد.

مثال ۲-۱۳: اعلان و پياده‌سازي رخدادها

```
using System;

public delegate void MyDelegate();

class Listing14-2
{
 public static event MyDelegate MyEvent;

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

  // فراخواني 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 ۲-۱۳ 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 فراخواني
  del();

  Console.ReadLine();
 }

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

باید توجه کنید که موارد کاربرد رخدادها بیشتر در برنامه‌هاي تحت ویندوز نمایان مي شود و در اینجا شاید وجود آنها در برنامه براي شما مشهود نباشد. در آینده، به بررسي برنامه‌نویسي فرمهاي ویندوز نیز خواهیم رسید و در آنجا به طور مفصل درباره event ها و delegate ها مجدداً بحث خواهیم نمود.

بطور خلاصه مي‌توان گفت، با استفاده از delegate ها روشي براي ایجاد دسترسي به متدها بط.ور پویا را فراهم نمودیم. با استفاده از رخدادها نیز، در صورت بروز اتفاقي خاص، عملي خاص انجام مي‌گیرد. این عمل معمولاً با

انجام مي‌گيرد.

توضيحات پيشرفته:

در انتهاي اين درس مي‌خواهم توضيحات پيشرفته تري را نيز در اختيار شما قرار دهم. در قسمت مربوط به delegate ها در همين درس، مثالي مطرح شد كه در آن delegate ي با نام Tick وجود داشت. اعلان اين delegate به صورت زير بود:

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

حال ميخواهيم به اين مثال يك رخداد نيز اضافه كنيم. در زير رخداد tick از نوع Tick اعلان شده است:

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

بايد توجه نماييد كه يك رخداد بطور خودكار ليست اعضاي خود را مديريت مي‌كند و نيازي به استفاده از يك مجموعه، مانند آرايه، براي مديريت اعضاي مرتبط با آن نيست.

نكته: يك رخداد بطور خودكار خود را تخصيص دهي مي‌كند و نيازي به ساخت نمونة جديد از روي يك رخداد وجود ندارد.

عضو شدن در يك رخداد (ثبت شدن در يك رخداد)

براي افزودن delegate جديد به يك رخداد كافيست تا از عملگر += استفاده نماييم. مثال زير كلاس Clock را نشان مي‌دهد كه در آن فيلدي از نوع Ticker با نام pulsed وجود دارد. كلاس Ticker داراي رخداد tick از نوع delegate ي بنام Tick است. متد Clock.Start، delegate ي از نوع Tick را با استفاده از عملگر += به pulsed.tick مي‌افزايد.

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

class Ticker
{
 public event Tick tick;
 ⋮
}

class Clock
{
 ⋮
 public void Start()
 {
```

```
 }
 :
 private void RefreshTime(int hours, int minutes, int seconds)
 {
  :
 }

 private Ticker pulsed = new Ticker();
}
```

هنگاميكه رخداد pulsed.tick اجرا مي‌شود، تمامي delegate هاي مرتبط با آن نيز فراخواني مي‌شوند كه در اينجا يكي از آنها RefreshTime است. (به مثال موجود در بخش delegate رجوع نماييد.)

خارج شدن از ليست يك رخداد

همانطور كه با استفاده از عملگر =+ مي‌توان delegate ي را به يك رخداد افزور، با استفاده از عملگر =- نيز مي‌توان delegate خاصي را از ليست اعضاي يك 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();
}
```

نكته: همانطور كه مي‌دانيد، عملگرهاي =+ و =- بر پاية دو عملگر اصلي + و - ايجاد شده‌اند. از اينرو در مورد delegate ها نيز مي‌توان از عملگر + استفاده نمود. استفاده از عملگر + براي delegate ها باعث ايجاد delegate جديدي مي‌شود كه به هنگام فراخواني هر دو delegate را به هم فرامي‌خواند.

فراخواني يك رخداد

يك رخداد نيز همانند delegate، با استفاده از دو پرانتز فراخواني مي‌گردد. پس از اينكه رخدادي فراخواني شد، كليه delegate هاي مرتبط با آن بترتيب فراخواني مي‌شوند. براي مثال در اينجا كلاس Ticker را در نظر بگيريد كه داراي متد private با نام Notify است كه رخداد tick را فرا مي‌خواند:

```
{
 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 چك کردن null نبودن رخداد tick ضروري است، چراکه فیلد رخداد بطور ضمني null در نظر گرفته مي‌شود و تنها زماني مقداري به غیر null میگیرد که delegate ي به آن مرتبط شده باشد. در صورت فراخواني رخداد null، خطاي NullReferenceException روي خواهد داد.

رخدادها داراي سطح امنیتي داخلي بسیار بالایي هستند. رخدادي که بصورت public اعلان مي‌شود، تنها از طریق متدها یا عناصر داخل همان کلاس قابل دسترسي است. بعنوان مثال، tick رخدادي درون کلاس Ticker است، از اینرو تنها متدهاي درون Ticker مي‌توانند tick را فرا بخوانند.

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

مثالي پیشرفته از استفادة رخدادها در فرمهاي ویندوز

حال که تا حدودي با رخدادها و ساختار آنها آشنا شدید، در این قسمت قصد دارم تا مقداري دربارة استفاده رخدادها در فرمهاي ویندوز و GUI ها صحبت نمایم. هر چند تا کنون کلیه برنامه‌ها و مطالبي که مشاهده کرده‌اید مبتني بر Console بوده‌اند، اما به علت استفاده بیشمار رخدادها در فرمهاي ویندوز و برنامه‌هاي مبتني بر GUI، لازم دیدم تا مطالبي نیز در این باره بیان کنم. هر چند فرمهاي ویندوز و GUI مطالبي هستند که خود نیاز به بحث و بررسي دقیق دارند و انشا ا... در رئوس آتي سایت مورد بررسي قرار خواهند گرفت. درصورتیکه مطالب این قسمت براي شما دشوار

یادگیري اين مطالب آنهم بدون مقدمه اندكي زود است، بيشتر هدف من از اين بخش آشنا شدن شما با كاربرد هاي پيشرفته‌تر رخدادها در برنامه‌نويسي بوده است.

كلاسهاي GUI مربوط به NET. Framework بطور گسترده‌اي از رخدادها استفاده مي‌نمايند. در مثالي كه در اينجا مورد بررسي قرار مي‌دهيم، برنامه‌اي وجود دارد كه داراي يك فرم به همراه دو دكمه (Button) بر روي آن است. اين دو دكمه بوسيلة دو فيلد از نوع Button ايجاد مي‌شوند. (Button عضو System.Windows.Forms است). كلاس Button از كلاس Control ارثبري مي‌كند و داراي رخدادي با نام Click از نوع EventHandler است. به مثال توجه نماييد.

```
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 نماييد كه كد فوق، كد مربوط به namespace مربوط به System است كه نحوة پيادەسازي آنرا نشان مي‌دهد. همانطور كه ملاحظه مي‌نماييد، درون System، delegate ي با نام EventHandler تعريف شده است. در زير اين namespace، اعلان System.Windows.Forms نيز آورده شده تا نحوة اعلان رخداد Click و ارثبري كلاس Button از كلاس Control نيز مشخص شود.

پس از اينكه بر روي دكمه‌اي واقع در فرم ويندوز كليك كنيد، Button بطور خودكار رخداد Click را فرا مي‌خواند.

براي كنترل اين رخداد ايجاد نمود. در مثالي كه در زير مشاهده مي‌كنيد، دكمه‌اي با نام Okay، متدي بنام Okay_Click و رخدادي 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.EventsArgs args)
 {
  :
 }
}
```

همانطور كه مشاهده مي‌كنيد، كلاس Example از System.Windows.Forms.Form مشتق مي‌شود، از اينرو تمامي خواص آن را به ارث مي‌برد. Okay نيز از نوع Button اعلان شده است. درون سازندة (Constructor) كلاس Example، متد Okay.Click به رخداد افزوده شده و مرجع this.Okay.Click نيز، متد مورد نظر را تعيين نموده است. همانطور كه گفته شد، EventHandler نيز delegate مورد نظر در اين مثال است. درون متد Okay_Click نيز مي‌توان كد خاصي را قرار داد تا عمل مورد نظر را انجام دهد. پس از كليك كردن بر روي دكمة Okay، عمل مورد نظري كه درون متد Okay_Click قرار داده شده، اجرا مي‌شود.

اين كد، شبيه به كدهايي است كه توسط محيطهاي برنامه‌سازي نظير Visual StudioNET. و يا C#Builder بطور خودكار توليد مي‌شوند و تنها كافيست تا شما كد مربوط به Okay_Click را درون آن وارد نماييد.

رخدادهايي كه توسط كلاسهاي GUI توليد مي‌شوند همواره از يك الگوي خاص پيروي مي‌كنند. اين رخدادها همواره از نوع delegate ي هستند كه مقدار بازگشتي ندارد (void) و داراي دو آرگومان است. آرگومان اول هميشه فرستندة رخداد و آرگومان دوم هميشه آرگومان EventArgs يا كلاس 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 BasicNET. 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.

پایان