آموزش گام به گام C#

www.tahlildadeh.com

مولف:مهندس افشین رفوآ

بسم الله الرحمن الرحیم

آموزشگاه تحلیل داده

تخصصی ترین مرکز برنامه نویسی و دیتابیس در ایران

کتاب آموزش گام به گام برنامه نویسی با C#

نویسنده : مهندس افشین رفوآ

تقدیم به نائب امام عصر، آیت الله خامنه ای

که عصا زدنش ضرب آهنگ حیدری دارد

تقدیم به همه جویندگان علم که توان و امکان شرکت در کلاس های حضوری ما را ندارند .

فهرست

مقدمه

زکات علم نشر آن است .حضرت علی (ع)

موسسه آموزشی تحلیل داده ، با حضور جمعی از متخصصین مجرب در زمینه برنامه نویسی در نظر دارد مطالب آموزشی خود را در قالب کتاب های آموزشی و فیلم ، به صورت رایگان در دسترس عموم قرار دهد تا حتی آن دسته از عزیزانی که بنا به دلایل مالی ،مسافت جغرافیایی و یا نداشتن وقت کافی ، امکان شرکت در دوره های حضوری برای آنها میسر نیست ، از یادگیری بی بهره نمانند .

علاوه ب راین علاقه مندان می توانند با ثبت نام در انجمن سایت تحلیل داده ، سوالات خود را مطرح نموده ومدرسین آموزشگاه و اعضای انجمن در اسرع وقت ، پاسخ های خود را حتی الامکان به صورت فیلم در دسترس عموم قرار دهند .

لذا از کلیه فعالان در این زمینه دعوت میشود در این حرکت جمعی در کنار ما باشند و با حضور فعال خود در انجمن گام موثری در بهبود سطح علمی جوانان کشور عزیزمان ایران بردارند .

آدرس سایت : http://www.tahlildadeh.com/

توجه :

برای دانلود سورس کد مثال های کتاب ، اینجا را کلیک کنید .

درس ۱ : معرفی آموزش زبان C#

معرفی آموزش زبان C#

به بخش جدید آموزش C# خوش آمدید. همزمان با معرفی چهارچوب کلری .NET ، مایکروسافت یک زبان برنامه نویسی جدید به نام C# که د ر اصطلاح سی شارپ خوانده می شود را به آن اضافه کرد.

C# طراحی شد تا نقش یک زبان برنامه نویسی ساده، مدرن، چند منظوره و شی گرا را برای چهارچوب کاری .NET. ایفا کند. از طرف دیگر، C# مفهوم های کلیدی و قابلیت های خوب سایر زبان های برنامه نویسی به ویژه جاوا را قرض کرده و درون خود دارد.

از لحاظ نظری، C# می تواند تا سطح کد ماشین یا اسمبلی کامپایل شود، اما در کارکرد واقعی، همیشه به همراه چهارچوب کاری .NET. استفاده می شود. بنابراین در برنامه ای که با زبان C# نوشته شده باشد، برای اجرا بر روی کامپیوتر، نیازمند نصب چهارچوب کاری .NET. می باشد. با وجود این که چهارچوب کاری .NET. امکان استفاده از طیف وسیعی از زبان های برنامه نویسی را بر روی ویندوز به ما می دهد، اما گاهی اوقات C# به عنوان زبان اصلی .NET. معرفی شده است. البته شاید این به دلیل طراحی همزمان با چهارچوب کاری .NET. باشد.

C# یک زبان برنامه نویسی شی گرا یا Oriented بوده و در آن امکان استفاده از متغیرها یا تابع سراسری یا Globol وجود ندارد. در C#، همه چیز در کلاس ها (closes) قرار می گیرند، حتی ساده ترین انواع داده ای مثل int یا String که از کلاس System.object مشتق شده اند.

در بخش های این مبحث آموزش C#، با مهم ترین موضوعات و کاربردهای C# آشنا خواهید شد.

درس ۲ : معرفی ابزار Visual C# Express

معرفی ابزار و آموزش کار با Visual C# Express:

دستورات زبان C# را می توانید با هر نوع ادیتور متنی مثل Windows Notepad نوشته و سپس به وسیله برنامه CSC.exe که کامپایلر خطی دستورات C# بوده و همراه با چهارچوب کاری .NET. بر روی سیستم نصب می شود، کامپایل و اجرا نمایید.

اما اکثر افراد ترجیح می دهند از یک IDE یا محیط توسعه یکپارچه (Integrated Development Envirment) استفاده کنند که مایکروسافت چندین برنامه را برای این منظور ارائه داده است. گل سرسبد این برنامه ها، Visual Studio است که قابلیت اجرای تمامی امکانات چهارچوب کاری .NET. را دارا می باشد. این برنامه بسیار پیشرفته بوده و در نسخه های مختلفی ارائه شده است.

از طرف دیگر ویژوال استودیو نسبتا نرم افزار گرانی بوده و برای استفاده برنامه نویسان معمولی مناسب نیست.

همزمان با ارائه نسخه ۲ چهار چوب کاری .NET.، مایکروسافت نسخه ای از ویژوال استودیو را به نام Express منتشر کرد که برای استفاده برنامه نویسان معمولی و کسانی که می خواهند چهارچوب کاری .NET. را فرا بگیرند، مناسب است. نسخه Express فقط برای برنامه نویسی به زبان های C# و VB.NET طراحی شده و برخی از قابلیت های مهم و کاربردی ویژوال استودیو را شامل نمی شود. اما به هر حال ابزار رایگانی بوده و برای افراد تازه کار و معمولی بسیار مناسب است.

برای برنامه نویسان به زبان C#، بایستی برنامه ویژوال C# اکسپرس را از آدرس +آدرس دانلود نموده و بر روی سیستم خود نصب نمایید. سپس آماده کد نویسی C# خواهید بود.

درس ۳ : آموزش ساخت اولین برنامه C# با Hello word

آموزش ساخت اولین برنامه C# با Hello word

اگر شما قبلا نیز اقدام به یادگیری یک زبان برنامه نویسی کرده باشید، حتما می دانید که اینگونه آموزش ها معمولا با یک برنامه ساده به نام "Hello word" شروع می شوند.

در این آموزش C# هم قصد داریم از این سنت قدیمی استفاده کنیم. برای این منظور برنامه Visual C# Express را اجرا کرده و مسیر منوی File - > New Project را طی کرده و گزینه Consule application را انتخاب نمایید. این ساده ترین نوع برنامه بر روی سیستم ویندوز است، اما نگران نباشید، ما خیلی اینجا نخواهیم ماند. پس از این که بر روی دکمه ok کلیک نمایید، برنامه Visual C# Express یک پروژه جدید را برای شما ایجاد می کند که حاوی یک فایل به نام Program.cs است.

این کار سرآغاز جایی است که هیجان کار شروع شده و کد آن بایستی به صورت زیر باشد :

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

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
```

درواقع، مجموعه کدهای فوق هیچ کار خاصی را انجام نمی دهند، یا حداقل اینگونه به نظر می رسند. سعی کنید با زدن دکمه F5 برنامه را بر روی سیستم خود اجرا نمایید. این کار باعث می شود تا Visual C# Express برنامه شامل را کامپایل کرده و اجرا نماید. اما همانگونه که مشاهده می کنید، این کدها کار خاصی را انجام نداده و فقط یک پنجره مشکی رنگ ویندوز باز شده و سپس بسته می شود. این مسئله به این دلیل است که برنامه شما دارای کد خاصی نبوده و کار خاصی را انجام نمی دهد. در بخش بعدی، به بررسی این خطوط کدها به طور کامل خواهیم پرداخت اما الان قصد داریم تا از برنامه خود یک خروجی بگیریم. برای این منظور دو خط کد زیر را درون آخرین مجموعه {} کد برنامه قرار دهید.

```
Console.WriteLine("Hello, world!");
Console.ReadLine();
```

سپس کد کامل برنامه بایستی به صورت زیر تغییر کند :

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

  namespace ConsoleApplication1
  {
    class Program
    {
      static void Main(string[] args)
      {
        Console.WriteLine("Hello, world!");
        Console.ReadLine();
      }
    }
  }
```

یک بار دیگر، جهت اجرای برنامه دکمه F5 را فشار دهید، این بار مشاهده خواهید کرد که پنجره سیاه برنامه باز شده و علاوه بر این که بلافاصله بسته نمی شود، بلکه یک پیام سلام "Hello word" را نیز به کاربر نشان می دهد. خب، ما دو خط کد به برنامه اضافه کردیم. اما این کدها در واقع چه کاری انجام می دهند؟

خط اول از کلاس Consale برای نمایش یک خط متن در خروجی استفاده کرده و خط دوم هم می تواند یک مقدار ورودی یا متن را از کاربر بخواند. اما چرا خواندن یا Read؟

درواقع این کار یک حقه کوچک است زیرا بدون آن برنامه بلافاصله اجرا شده و تمام می شود و بدون این که کاربر فرصت کند خروجی آن را بروی صفحه ببیند، پنجره اش بسته می شود.

دستور ReadLine Command به برنامه می گوید تا برای دریافت یک ورودی از کاربر صبر کند و همان طور که مشاهده می منید، شما می توانید یک متن را درون پنجره تایپ نمایید. پس از تایپ متن دلخواه، دکمه Enter را زده و پنجره برنامه را ببندید.

به شما تبریک می گوییم، اولین برنامه C# خود را ساخته و اجرا کردید. در بخش بعدی به تشریح کدهای نوشته شده و عملیات صورت گرفته خواهیم پرداخت.

درس ۴ : توضیح و آموزش برنامه Hello Word در C#

در بخش قبل، در اولین برنامه C# ای که طراحی کردیم، یک نوشته را در خروجی چاپ نمودیم. برای درک بستر خروجی مثال، در درس قبلی به تشریح کدهای نوشته شده پرداخته نشد، اما در این درس به بررسی کد مثال می پردازیم.

همانطور که از مشاهده کدهای مثال درس متوجه شده اید، برخی از خط های کد مثال بسیار شبیه هم بودند، بنابراین در توضیح، آن ها را در کنار همدیگر قرار داده ایم.

بیایید با کوتاه ترین و پرکاربرد ترین کاراکترها در کد مثال خود شروع کنیم : کاراکترهای { و }. به این کاراکترها در اصطلاح براکت (curly braces) می گویند و در C#، ابتدا و انتهای هر بلوک کد را مشخص می کنند. براکت ها در بسیاری از زبان برنامه نویسی دیگر از جمله C++، جاوا، جاوا اسکریپت و ... نیز استفاده می شوند. همانطور که در مثال مشاهده کردید، براکت ها برای بسته بندی چندین خط کد که مرتبط به هم هستند، استفاده می شوند. در مثال های بعدی، با نحوه استفاده از براکت ها بیشتر آشنا خواهید شد.

اکنون بیایید از ابتدای کد شروع کنیم. قسمت Using ها :

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

Using یک واژه کلیدی (keyword) است که توسط ادیتورکدها به رنگ آبی نشان داده می شود. واژه کلیدی Using یک namespace را به کد برنامه وارد می کند. Namespace مجموعه ای از کلاس ها هستند که با هم کار خاصی را انجام می دهند. در مثال Hello Word، سه namespace به برنامه اضافه شده اند که هر کدام کار خاصی را در کد برای ما انجام می دهند. برای مثال، ما از کلاس Consale Class که بخشی از System Namespace است برای چاپ مقدار متنی در خروجی استفاده می کنیم.

از طرف دیگر، همانند قطعه کد زیر، شما می توانید یک namespace دلخواه را ایجاد کرده و سپس آن را در کدهای خود استفاده کنید.

```
namespace ConsoleApplication1
```

اکنون namespace ConsaleAPP، به عنوان namespace اصلی (main) این برنامه بوده و شامل چندین کلاس خواهد بود. می توانید چندین namespace دیگر نیز که نیاز دارید را ایجاد کرده و در هر کدام کلاس های مورد نظر خود را قرار دهید. سپس همانند سایر namespace ها با استفاده از واژه کلیدی Using آن را به برنامه خود اضافه کنید.

در مرحله بعد، ما کلاس Closs مورد نظر خود را ایجاد می کنیم. از آن جا که C# یک زبان شی گرا Object-oriented است، مجموعه کدهای خاص را درون یک کلاس تعریف می کنیم. برای مثال، به وسیله کد زیر یک کلاس به نام Program را تعریف کرده ایم :

```
class Program
```

در هر برنامه به تعداد نیاز می توانید کلاس های مختلف تعریف کنید. اما در این مثال، ما فقط یک کلاس خواهیم داشت. هر کلاس می تواند شامل تعدادی متغیر (variable)، خواص (properties) و متدها (methods) باشد، مفهوم هایی که در درس های بعد به تشریح کامل آن ها خواهیم پرداخت. تنها چیزی که الان بایستی

بدانید این است که کلاس Program فقط شامل یک متد (method) بوده که به صورت زیر تعریف شده است :

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

خط کد فوق، به احتمال زیاد پیچیده ترین بخش مثال است. به همین دلیل بیایید ان را به بخش های کوچکی تقسیم کرده و به بررسی هریک بپردازیم.

کلمه اول static است. کلیدواژه یا (keyword) به نام static اعلام می کند که این کلاس بایستی بدون نیاز به نمونه سازی از آن قابل دسترس باشد، راجع به این مسئله در بخش Classes به طور کامل توضیح می دهیم.

واژه کلیدی بعدی Void است که مشخص می کند آیا متد ما بایستی پس از اجرای کامل، مقداری را برگرداند یا خیر. مقدار برگشتی یک تابع می تواند برای مثال از نوع عددی int، متنی string و یا هیچ چیز (void) باشد. به کار بردن کلمه void در این مثال، به این معنی است که تابع ما هیچ مقداری را پس از اجرا باز نمی گرداند.

واژه کلیدی بعدی Main بوده که نام متد را تعیین می کند. متد Main تابع اصلی هر برنامه C# بوده و اولین قطعه کدی است که در برنامه اجرا می شود.

پس از نام متد، می توانید یک یا چند آرگومان را به عنوان مقادیر ورودی هر متد درون پرانتز تعیین کنید. در مثال ما، متد Main فقط دارای یک آرگومان ورودی به نام args می باشد. آرگومان یک متغیر یا مقدار ثابت است که در هنگام فراخوانی هر تابعی، می تواند به آن ارسال شود. آرگومان args در مثال فوق از نوع متنی و آرایه ای است.

در پایان این درس بایستی درک کلی از برنامه های C# و نحوه کار آن ها پیدا کرده باشید.

درس ۵ : آموزش انواع داده ای Data Types در C#

آشنایی با انواع داده ای Data Types در C# :

انواع داده ای یا Data Types در تمامی قسمت های یک زبان برنامه نویسی مثل C# استفاده می شود. به دلیل این که زبان C#، یک زبان قدرتمند داده ای است، بایستی هر زمان که یک متغیر را تعریف و استفاده می کنید، به کامپایلر اطلاع دهید آن متغیر از چه نوع داده ای است. به نحوه و تعیین نوع داده ای یک متغیر به طور کامل در بخش متغیرها (Variable) خواهیم پرداخت. در این درس به طور کلی به بررسی انواع داده ای مهم زبان C# و نحوه کارکرد آن ها می پردازیم.

نوع داده ای bool ساده ترین Data Type زبان C# است. این نوع داده ای که به نوع درست یا غلط نیز معروف است، فقط دو نوع مقدار می تواند داشته باشد، true یا false. متغیر bool در هنگام استفاده از عملگرهای منطقی و یا دستورات شرطی مثل if بسیار کاربرد دارد.

انواع داده ای int نیز که مخفف کلمه integer است، برای نگهداری اعداد بدون بخش اعشاری آن ها به کار می رود. نوع داده ای int پرکاربردترین متغیر در هنگام کار با اعداد در C# است. متغیرهای integer، بر حسب اندازه عددی که می توانند نگهداری کنند، دارای انواع مختلفی در زبان C# می باشند.

نوع داده ای String نیز برای نگهداری متن یا text به کار می رود که عبارت است از تعدادی کاراکتر پشت سر هم. در C#، متغیرهای String از نوع immutable یا تغییرناپذیر هستند، به این معنی که متغیرهای String، پس از تعریف و مقداردهی هرگز تغییر نمی کنند. هنگام کار با متدهایی که یک متغیر String را دستکاری و تغییر می دهند، متغیر String اول تغییر نمی کند، بلکه یک متغیر جدید با مقدار جدید ایجاد می شود.

متغیر char نیز برای نگهداری یک کاراکتر تنها به کار می رود.

متغیر float نیز برای نگهداری اعداد اعشاری استفاده می شود.

درس ۶ : آموزش تعریف و مقداردهی متغیرها Variable در C#

آموزش تعریف و مقداردهی متغیرها Variable در C# :

یک متغیر یا Variable بخشی از حافظه سیستم است که همانند یک اتاق برای نگهداری اطلاعات خاصی به کار می رود. متغیرها اساس کار برنامه های C# بوده و به صورت زیر، قابل تعریف هستند :

```
<data type> <name>;
```

به عنوان مثال در کد زیر یک متغیر از نوع String به نام name را تعریف کرده ایم :

```
String name;
```

در کد فوق <data type> ، نوع داده ای متغیر و <name> نام آن را تعیین می کند.

حالت فوق، ساده ترین حالت تعریف یک متغیر است. اما ممکن است شما بخواهید میدان دید یا Visibility خاصی را برای متغیر خود تعیین کرده و در هنگام تعریف، آن را متد دهی نیز کنید. این کار بایستی به صورت زیر انجام شود :

```
<visibility>  <data type>  <name>  = <value>;
```

در نمونه فوق <visibility> میدان دید متغیر را تعیین می کند. یعنی می گویند چه توابع، کلاس ها و یا متغیرهای دیگری در سطح برنامه می توانند این متغیر را ببینند، فراخوانی و دستکاری کنند. <value> نیز که مقدار اولیه متغیر را مشخص می کند. کد زیر یک مثال را برای تعریف کامل متغیر در C# نشان می دهد :

```
Private String name = “Tahlil adeh”;
```

در کد مثال زیر، نحوه تعریف، مقداردهی و کار با چند متغیر در زبان C# را در عمل نشان داده ایم :

```
using System;

namespace ConsoleApplication1
{
```

```
class Program
{
    static void Main(string[] args)
    {
        string firstName = "John";
        string lastName = "Doe";

        Console.WriteLine("Name: " + firstName + " " + lastName);

        Console.WriteLine("Please enter a new first name:");
        firstName = Console.ReadLine();

        Console.WriteLine("New name: " + firstName + " " + lastName);

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

بسیار خب، بخش زیادی از کد مثال فوق را قبلا توضیح داده ایم، بنابراین در این مرحله مستقیم به سراغ بخش مورد نظرمان می رویم.

اول از همه، ما چندین متغیر از نوع String Type را تعریف کرده ایم. یک String می تواند شامل متن یا Text باشد و همانطور که در کد مثال می بینید، هر متغیر String را بلافاصله مقدار دهی کرده ایم. سپس یک خط متن

را به وسیله دو متغیر بر روی خروجی Consale نشان داده ایم. از کاراکتر (+) برای چسباندن متن دو متغیر String و نمایش آن ها به صورت یک جمله استفاده شده است.

در مرحله بعدی، از کاربر خواسته ایم تا یک مقدار جدید را برای متغیر firstName وارد کند. برای این منظور از دستور ReadLine() استفاده شده که یک مقدار ورودی را از کاربر دریافت کرده و درون متغیر firstName قرار می دهد. پس از این که کاربر نام مورد نظر خود را وارد کند، مقدار جدید در متغیر firstName وارد شده و سپس مجددا به وسیله دستور Consale.WriteLine()، جمله را با نام جدید به کاربر نشان داده ایم.

در کد فوق ما فقط از یک متغیر استفاده کرده ایم و این کد به خوبی مهم ترین قابلیت یک متغیر یعنی تغییر در هنگام اجرای برنامه (Run Time) را نشان می دهد.

مثال بعدی برای نشان دادن کار متغیرها، انجام عملیات ریاضی است. کد زیر نحوه انجام کار را نشان می دهد :

```
int number1, number2;

Console.WriteLine("Please enter a number:");
number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Thank you. One more:");
number2 = int.Parse(Console.ReadLine());

Console.WriteLine("Adding the two numbers: " + (number1 + number2));

Console.ReadLine();
```

کد فوق را در تابع Main مثال قبل قرار داده و برنامه را مجددا اجرا کنید. تنها حقه به کار رفته در مثال دوم، استفاده از متد int.Parse() است. این متد یک مقدار متنی String را خوانده و آن را به یک متغیر عددی integer تبدیل می کند.

همانطور که مشاهده می کنید، در کد مثال دوم برنامه هیچ تلاشی برای اعتبارسنجی (validate) مقدار ورودی کاربر انجام نداده و اگر کاربر یک مقدار رشته ای (متن) را وارد کرده و دکمه Enter را بزند، برنامه با اشکال رو به رو خواهد شد، زیرا عمل ریاضی را بر روی متن نمی تواند انجام دهد. برای حل این مشکل در درس های بعد بیشتر توضیح می دهیم.

درس ۷ : آموزش ساختار دستوری if در C#

آموزش ساختار دستوری if در C# :

یکی از مهم ترین ساختارهای دستوری در هر زبان برنامه نویسی از جمله C#، دستور if است. توانایی ساخت دستورات شرطی کلی از مهم ترین کارهایی است که بایستی بتوان با زبان ها ی برنامه نویسی انجام داد. در C#، ساختار شرطی if بسیار ساده و کاربردی است. اگر از شاختار دستوری شرطی if در هر زبان برنامه نویسی دیگری استفاده کرده باشید، به راحتی می توانید در C# نیز از آن بهره بگیرید. ساختار دستور شرطی if نیاز مند یک مقدار boolean است که یا true است یا false. در برخی از زبان های برنامه نویسی، چندین نوع داده ای را می توان به صورت اتوماتیک به Boolean تبدیل کرد، اما در C# باید مستقیما از نوع داده ای Boolean استفاده کنید. برای مثال نمی توانید از نوع داده ای عددی int به صورت مستقیم استفاده کنید، اما می توانید آن را با یک مقدار دیگر قیاس کرده و مقدار true یا false نتیجه را مورد استفاده قرار دهید.

در درس قبلی، به آموزش نحوه تعریف و استفاده از متغیرها (Variables) در زبان C# پرداختیم. در این درس هم به آموزش نحوه کار با دستورات شرطی در C# می پردازیم. برای این منظور کد مثال زیر را با دقت مطالعه کنید

```
using System;
```

```
namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      int number;

      Console.WriteLine("Please enter a number between 0 and 10:");
      number = int.Parse(Console.ReadLine());

      if (number > 10)
        Console.WriteLine("Hey! The number should be 10 or less!");
      else
        if (number < 0)
          Console.WriteLine("Hey! The number should be 0 or more!");
        else
          Console.WriteLine("Good job!");

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

در کد مثال فوق از دو دستور شرطی if استفاده کرده ایم تا چک کنیم آیا عدد ورودی توسط کاربر، بین ۰ تا ۱۰ است یا خیر. به همراه دستور if از کلید واژه else نیز استفاده شده است. قسمت کد else زمانی اجرا می شود که شرط بخش if درست نباشد.

اگر در کد مثال فوق دقت کرده باشید، ما از کاراکترهای {} برای دسته بندی دستورات if و else ها استفاده نکرده ایم.. این یک قانون در C# است. اگر کد نوشته شده فقط در حد یک متن خطی باشد، نیازی به استفاده از {} برای دسته بندی کدها نیست.

حجم کد نوشته شده برای مقایسه مقدار یک عدد در مثال فوق کمی زیاد به نظر می رسد. همان کد را می توانید به صورت زیر، خلاصه تر بنویسید :

```
if((number > 10) || (number < 0))
   Console.WriteLine("Hey! The number should be 0 or more and 10 or less!");
else
   Console.WriteLine("Good job!");
```

ما هر یک از دستورات شرطی را درون یک پرانتز قرار داده و سپس از عملگر || که به معنلی "یا"، "or" می باشد، استفاده کرده ایم تا چک کنیم عدد واردشده از ۱۰ بزرگتر یا کوچکتر از ۰ است. عملگر دیگری که در این گونه موارد نیز می توانید استفاده کنید عملگر AND به معنای "و" است که به صورت && نوشته می شود. آیا می توانیم در کد مثال دوم از عملگر AND به جای or استفاده کنیم؟. بله، فقط کافی است کد را به صورت زیر بازنویسی کنیم :

```
if((number <= 10) && (number >= 0))
   Console.WriteLine("Good job!");
else
   Console.WriteLine("Hey! The number should be 0 or more and 10 or less!");
```

در این درس عملگرهای "کوچکتر از" و "بزرگتر از" را نیز معرفی کردیم.

درس ۸ : آموزش کار با دستور Switch در C#

آموزش کار با دستور Switch در C# :

دستور Switch همانند مجموعه ای از دستورات پشت سر هم if عمل می کند. دستور Switch درواقع لیستی از حالت های ممکن است که برای هر حالت یک دستور یا کدی جهت اجرا پیش بینی شده است. این دستور همچنین یک حالت default یا پیش فرضی دارد که در صورتی که هیچ یک از حالات آن true نباشد، اجرا خواهد شد.

یک دستور ساده Switch ساختاری همانند کد مثال زیر دارد :

```
int number = 1; switch(number) {     case 0:         Console.WriteLine("The number is zero!");
break;    case 1:      Console.WriteLine("The number is one!");      break; }
```

مقدار شناسه یا identifier (مقدار متغیری که می خواهیم دستور Switch بر حسب آن عمل کند) را پس از واژه کلیدی Switch قرار می دهیم. سپس لیستی از انواع حالت های مختلف برای آن مقدار توسط هر دستور case مشخص شده و مقدار شناسه با مقدار هر case مقایسه می شود. اگر مقدار شناسه با مقدار هر کدام برابر نبود، دستورات بخش پیش فرض default اجرا می شود.

اگر دقت کرده باشید، در پایان هر case یک دستور break قرار داده شده است. برای چیست؟ اگر مقدار یک case با مقدار شناسه دستور Switch برابر شود، دستورات آن case اجرا شده و اگر در انتهای case دستور break وجود نداشته باشد، دستورات case های بعدی نیز اجرا خواهد شد. به عبارت دیگر، دستور break ، باعث توقف روند اجرای دستورات Switch شده و برنامه به خط کد بعد از Switch می فرستد. در نوشتن break دقت لازم به عمل آورید، ؟؟؟؟ نوشتن آن می تواند کلا برنامه را دچار اختلال کنید.

هنگامی که هم یک تابع را در دستورات یک case تعیین می کنید، می توانید با استفاده از return خروجی تابع را به عنوان خروجی Switch برگردانید.

در کد مثال، ما از یک متغیر عددی integer به عنوان شناسه دستور Switch استفاده کرده ایم، اما این شناسه می تواند از نوع متنی String و یا هر نوع داده ای دیگر زبان C# باشد.

در مثال زیر، ابتدا یک مقدار را به عنوان ورودی از کار برگررفته ایم، سپس آن را به عنوان یک متغیر متنی String به دستور Switch ارسال نموده ایم. در کد مثال زیر، برای هر دو مقدار "yes" و "Maybe" یک دستور مشترک را تعیین کرده ایم.

```
Console.WriteLine("Do you enjoy C# ? (yes/no/maybe)");
string input = Console.ReadLine();
switch(input.ToLower())
{
  case "yes":
  case "maybe":
    Console.WriteLine("Great!");
    break;
  case "no":
    Console.WriteLine("Too bad!");
    break;
}
```

در کد مثال فوق، سوالی از کاربر پرسیده شده و از وی خواسته شد یکی از مقادیر "yes"، "no" و "maybe" را وارد نماید. سپس مقدار وارد شده از ورودی توسط دستور Consale.ReadLine() خوانده شده و یک دستور Switch بر مبنای آن طراحی شده است. برای سهولت کار کاربر، توسط دستور ToLower() ، کاراکترهای ورودی کاربر را به حروف کوچک تبدیل کرده ایم تا در هنگام مقایسه با مقادیر Case ها، فرقی بین مقدار وارد شده برای حروف بزرگ و کوچک وجود نداشته باشد.

اما بدون وجود دستور default در Switch، اگر کاربر در مثال فوق مقداری غیر از مقادیر تعیین شده برای case ها وارد نماید، برنامه هیچ خروجی نخواهد داشت. بنابراین کد مثال فوق را به همراه default به صورت زیر بازنویسی می کنیم :

```
Console.WriteLine("Do you enjoy C# ? (yes/no/maybe)");
string input = Console.ReadLine();
switch(input.ToLower())
{
  case "yes":
  case "maybe":
    Console.WriteLine("Great!");
    break;
  case "no":
    Console.WriteLine("Too bad!");
    break;
  default:
    Console.WriteLine("I'm sorry, I don't understand that!");
    break;
}
```

اگر کاربر مقداری به غیر از مقادیر تعیین شده برای case ها وارد نماید، بخش default به صورت پیش فرض اجرا خواهد شد.

درس ۹ : آموزش کار با حلقه ها Loops در C#

آموزش کار با ساختارهای تکرار حلقه Loops در C#

یکی دیگر از تکنیک های اصلی در زمان نوشتن نرم افزارها، امکان ایجاد حلقه ها دستوری یا looping است. این نوع دستورات امکان تکرار بلوک هایی از کد برای دفعات دلخواه را می دهند. برای مثال ما می خواهیم گروهی از دستورات تا زمانی که مقدار یک متغیر مثلا کمتر از ۱۰ است، تکرار شود، در این حالت بایستی از حلقه ها در C# استفاده کنیم.

در زبان C#، چهار نوع حلقه اصلی داریم که در ادامه به تشریح هر یک از آن ها با ارائه مثال های عملی خواهیم پرداخت :

حلقه While loop :

حلقه While loop آسان ترین نوع حلقه در زبان C# بوده و به همین دلیل آموزش حلقه ها را از این حلقه شروع می کنیم. حلقه While loop مجموعه بلوک دستورات تعیین شده برای آن را تا زمانی که شرط تعیین شده برای حلقه درست true باشد، اجرا می کند. کد زیر یک مثال ساده از کاربرد حلقه While را نشان می دهد، در ادامه به تشریح بیشتر کد می پردازیم :

```
using System;

 namespace ConsoleApplication1
 {
   class Program
   {
     static void Main(string[] args)
     {
       int number = 0;
```

```
        while (number < 5)
        {
          Console.WriteLine(number);
          number = number + 1;
        }

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

برنامه را اجرا نمایید. لیستی از اعداد ۰ تا ۴ برای شما نمایش داده خواهد شد. متغیر number که درواقع شمارنده حلقه While مثال نیز هست، از عدد ۰ شروع شده و هر بار که دستورات حلقه یک بار اجرا می شوند، مقدار آن یک واحد افزایش پیدا می کند.

تا زمانی که مقدار متغیر number کمتر از ۵ و نه خود ۵ باشد، اجرای دستورات ادامه پیدا کرده و با رسیدن عدد number به ۵، اجرای حلقه متوقف شده و برنامه به خط کد بعد از حلقه While پرش می کند.

حلقه do loop:

کارکرد حلقه do loop کمی متفاوت با حلقه While loop است. در حلقه do loop ابتدا دستورات بدنه حلقه یک بار اجرا شده و در پایان شرط حلقه چک می شود. در صورت درست بودن شرط، باز هم دستورات حلقه تکرار می شود.

نکته مهم در مورد حلقه do loop این است که اگر حتی شرط حلقه از ابتدا درست نباشد، دستورات حلقه حداقل یک بار اجرا می شوند.

کد زیر یک مثال عملی از کار با حلقه do loop را نشان می دهد :

```
do
```

```
{
   Console.WriteLine(number);
   number = number + 1;
} while(number < 5);
```

خروجی حلقه do loop مثال فوق اعداد ۰ تا ۵ را چاپ کرده و با رسیدن شماره حلقه به ۶، اجرای آن متوقف می شود.

حلقه for loop :

مکانیزم حلقه for loop با حلقه های قبلی کمی متفاوت است. از این حلقه بهتر است زمانی استفاده شود که می دانیم حدودا حلقه چند بار ممکن است تکرار شود. حلقه for loop دارای یک عدد به عنوان شمارنده است که با هر بار اجرای حلقه، مقدار آن به اندازه واحد تعیین شده کم یا زیاد می شود. اجرای حلقه تا زمانی که شرط آن درست باشد، ادامه داشته و شرط در ابتدای اجرای حلقه تست می شود. کد زیر یک مثال عملی از حلقه for loop را نشان می دهد :

```
using System;

  namespace ConsoleApplication1
  {
    class Program
    {
      static void Main(string[] args)
      {
        int number = 5;

        for (int i = 0; i < number; i++)
          Console.WriteLine(i);
```

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

کد مثال فوق خروجی همانند حلقه While loop ایجاد می کند اما همانطور که شماهده می کنید، این حلقه کمی متراکم تر از حلقه های قبلی است. حلقه for از سه بخش اصلی تشکیل شده است :

ابتدا یک مقدار متغیر را برای شمارش اجرای حلقه تعریف و مقداردهی می کنیم، یک شرط برای اجرای حلقه بر مبنای متغیر شمارنده تعیین شده و یک گام افزایش یا کاهش جهت شمارش اجرای حلقه و کنترل گام آن مثل ++ یا – تعیین می گردد.

در کد مثال فوق، در مرحله اول شمارنده حلقه را به نام i تعریف و با مقدار ۰ مقداردهی کرده ایم، این بخش حلقه فقط یک بار و در هنگام شروع حلقه اجرا می شود.

دو قسمت بعدی حلقه، در هر بار تکرار حلقه، اجرا می شوند. در هر بار، مقدار متغیر i با متغیرهای عددی number مقایسه شده و در صورتی که کوچکتر از آن باشد، دستورات حلقه یک بار دیگر اجرا شده ومقدار i یک واحد افزایش می یابد.

نکته مهم : اگر برای متغیر حلقه ، گام افزایش یا کاهش تعیین نکنید، در صورت درست بودن اولیه شرط حلقه، حلقه for به صورت نامحدود اجرا می شود.

نکته ۲ : گام افزایش یا کاهش شمارنده حلقه را به جای پرانتز جلوی for، در درون بدنه دستورات حلقه نیز می توان قرار داد.

حلقه foreach loop :

حلقه آخری که در این بخش بررسی خواهیم کرد، حلقه foreach loop است. از این حلقه معمولا در هنگام کار با مجموعه ای از آیتم ها مثل آرایه ها (Arrays) و یا متغیرهای لیستی استفاده می شود. در کد مثال حلقه foreach از یک متغیر لیستی به نام ArrayList استفاده می کنیم.

به کد مثال عملی حلقه foreach loop دقت کنید :

```
using System;
  using System.Collections;

  namespace ConsoleApplication1
  {
    class Program
    {
      static void Main(string[] args)
      {
        ArrayList list = new ArrayList();
        list.Add("John Doe");
        list.Add("Jane Doe");
        list.Add("Someone Else");

        foreach (string name in list)
          Console.WriteLine(name);

        Console.ReadLine();
      }
    }
```

}

در ابتدای مثال، ابتدا یک نمونه از متغیر لیستی ArrayList به نام list را ایجاد و سپس چندین مقدار متنی را به عنوان اعضای آن، به آرایه خود اضافه کرده ایم.

با استفاده از حلقه foreach loop به درون هر یک از آیتم های آرایه رفته و مقدار آن عضو را برابر با متغیر name قرار داده ایم. سپس مقدار متغیر name را در خروجی چاپ کرده ایم.

همانطور که در کد مثال مشاهده می کنید، ما به حلقه foreach اعلام کرده ایم که متغیر name از نوع متنی string می باشد. در هنگام کار با حلقه foreach حتما بایستی نوع داده ای که می خواهید حلقه درون آن حرکت کند را اعلام کنید.

در مواردی که نوع داده ای اعضای آرایه شما ممکن است با هم متفاوت باشد، بهتر است از یک متغیر شی کلاس object class برای حرکت درون اعضای آرایه استفاده کنید.

نکته : در هنگام کار با متغیرهای آرایه ای، حتما از حلقه foreach استفاده کنید. زیرا ساختار بسیار ساده تری نسبت به سایر حلقه ها در زبان C# دارد.

درس ۱۰ : آموزش کار با توابع Function در زبان C#

آموزش کار با توابع Function در زبان C# :

یک تابع یا Function در C#، به شما امکان می دهد مجموعه ای از دستورات را درون یک ظرف مشخص قرار داده و در هر جای برنامه که لازم داشتید، با به کار بردن نام تابع آن ها را فراخوانی کنید.

در برنامه نویسی پروژه ها، شما گاهی مجبور می شوید تا یک قطعه کد را در چندین محل مختلف مورد استفاده قرار دهید، در این حالت است که تابع ها (Function) به کمک شما می آیند. از طرف دیگر، به وسیله توابع شما می توانید کدهای پروژه خود را به بخش های مجزا و قابل تفکیک از هم تبدیل کنید.

در زبان C#، یک تابع (Function) به صورت کلی زیر تعریف می شود :

```
<visibility><return type><name>(<parameters>)
{
        <function code>
}
```

برای فراخوانی یک تابع، کافی است نام آن را نوشته و سپس یک پرانتز باز و بسته در مقابل آن قرار دهید. اگر تابع شما دارای یک یا چند پارامتر باشد، لیست پارامترها را هم در همین پرانتز قرار می دهید. به صورت زیر :

```
FunctionName (Parameter1, Parameter2,...);
```

در کد مثال زیر یک تابع به نام Dostuff را فراخوانی کرده ایم :

```
public void DoStuff()
{
  Console.WriteLine("I'm doing something...");
}
```

در کد تابع فوق، اولین بخش فراخوانی تابع، یعنی کلمه public، تعیین کننده میدان دید تابع در سطح برنامه بوده و تعیین آن اختیاری است. میدان دید یک تابع مشخص می کند آیا سایر توابع و کلاس های موجود در برنامه امکان مشاهده و دسترسی تابع مورد نظر را خواهند داشت یا خیر. Public به معنای عمومی بوده و یعنی سایر کلاس ها و توابع دیگر برنامه می توانند به تابع فوق دسترسی داشته باشند.

نکته مهم : اگر میدان دید برای یک تابع تعیین نشود، به صورت پیش فرض Private یا خصوصی در نظر گرفته می شود. Private تابع های هم کلاس تابع مورد نظر امکان دسترسی مستقیم به تابع را دارند.

در درس های بعدی به طور کامل به بررسی میدان دید یا Scope توابع خواهیم پرداخت.

بخش بعدی قسمت فراخوانی تابع تعیین نوع داده ای مقدار خروجی تابع است. این مقدار می تواند هر نوع داده ای خاص در C# بوده و یا Void تعیین شود. به کار بردن کلمه Void به این معناست که این تابع هیچ مقدار خاصی را بر نمی گرداند.

از طرف دیگر همانطور که می بینید پرانتزهای مقابل نام تابع خالی هستند و به این معناست که این تابع هیچ پارامتری را دریافت نمی کند.

بیایید کمی تابع فوق را به صورت زیر تغییر دهیم :

```
public int AddNumbers(int number1, int number2)
{
   int result = number1 + number2;
   return result;
}
```

در کد جدید تقریبا همه بخش ها را تغییر دادیم. تابع اکنون یک مقدار عددی integer را به عنوان خروجی برمی گرداند، در پارامتر عددی integer دریافت کرده و به جای چاپ متن در خروجی، یک عملیات ریاضی انجام داده و خروجی آن را به عنوان متغیر result بر می گرداند.

این کار به این معناست که هر موقع نیاز داشتیم، در هر بخشی از کد برنامه می توانیم تابع فوق را فراخوانی کرده و با ارسال دو عدد مورد نظر به آن، حاصل جمع آن ها را به صورت خروجی دریافت کنیم، تا این که هر بار لازم باشد کد عملیات ریاضی را مجددا بنویسیم.

تابع AddNumber() را می توانیم به راحتی توسط کد زیر در هر جای برنامه فراخوانی کرد ه و در حجم و زمان کدنویسی خود صرفه جویی زیادی بکنیم.

```
int result = AddNumbers(10, 5);
Console.WriteLine(result);
```

همانطور که اشاره کردیم، این تابع یک مقدار عددی را باز می گرداند. هنگامی که در یک تابع، هر نوع داده ای غیر از Void را تعیین می کنیم، به این معناست که تابع را مجبور نموده ایم تا یک مقدار برگشتی داشته باشد.

نکته : اگر خط کد return را از کد مثال فوق بردارید، خواهی دید که در هنگام اجرای برنامه، کامپایلر بر بروی این تابع ارور داده و پیام زیر را صادر می کند :

'AddNumbers(int, int)': not all code paths return a value

پیام فوق به ما یادآوری می کند که تابع فوق، علی رغم این که یک مقدار خروجی برای آن در نظر گرفته شده، اما هیچ خروجی را بر نمی گرداند. اما می خواهیم خروجی تابع را کنترل کنید، می توانید کدی مثل کد زیر را به تابع اضافه کنید :

```
public int AddNumbers(int number1, int number2)
  {
    int result = number1 + number2;
    if (result > 10)
    {
      return result;
    }
  }
```

اما باز هم پیام خطایی همانند مثالق قبل صادر می شود، چرا؟ به این دلیل که هیچ ضمانتی وجود نداشته که شرط دستور if درست از آب در بیاید و برنامه خروجی داشته باشد (دستور return اجرا بشود). می توانید مشکل فوق را با تعیین یک مقدار پیش فرض برای عبارت return به صورت زیر حل کنید :

```
public int AddNumbers(int number1, int number2)
  {
```

```
    int result = number1 + number2;
    if (result > 10)
    {
      return result;
    }
    return 0;
}
```

کد اضافه شده فوق، مشکل برنامه ما را حل کرده و از طرف دیگر نشان می دهد می توانیم بیش از یک دستور return را در بدنه تابع خود تعریف کنیم. به محض اجرای دستور return در هر جای کد برنامه، خروجی تابع ارسال شده و اجرای مابقی دستورات تابع لغو می شود. در مثال فوق، اگر مقدار متغیر خروجی return بزرگتر از ۱۰ باشد، هیچ گاه دستور return 0; اجرا نمی شود.

درس ۱۱ : آموزش کار با پارامترهای تابع در C#

آموزش کار با پارامترهای تابع در زبان C# :

در درس قبلی، به طور کامل نحوه کار با تابع ها در C# و فراخوانی و استفاده آن ها را آموزش دادیم. تا حدودی به آموزش نحوه تعیین و ارسال پارامترهای تابع هم پرداختیم. اما در این درس به صورت اختصاصی قابلیت ها و کارکردهای پارامترها را بررسی خواهیم کرد.

اولین چیزی که به بررسی آن خواهیم پرداخت، تغییردهنده های ref و out یا modifiers هستند. زبان C# همانند اکثر زبان های برنامه نویسی دیگر، بین متغیرهای ارسال با مقدار "by value" و ارسال با رفرنس "by refrence" تفاوت قائل می شود.

حالت پیش فرض در زبان C#، ارسال با مقدار یا "by value" است. این حالت به معنای آن است که وقتی شما یک متغیر را به تابع ای به عنوان پارامتر ارسال می کنید، درواقع یک کپی از آن شی را می فرستید و نه رفرنس به آدرس عمل واقعی متغیر. از طرف دیگر این کار به این معنی است که شما می توانید تغییرات و عملیات مورد

نظر خود را بر روی متغیر پارامتر ارسالی انجام دهید. بدون این که شی اصلی آن متغیر را در حافظه دستکاری نمایید و به عبارت دیگر هر کاری بر روی پارامتر در درون تابع انجام می دهید، بر روی نسخه کپی آن اجرا می شود نه متغیر اصلی.

با استفاده از واژه های کلیدی ref و out می توانید عملکرد فوق را تغییر داده و نسخه اصلی یک متغیر را به جای مقدار آن به تابع ارسال نمایید.

آموزش ref modifier :

کد مثال زیر را مطالعه نمایید :

```
static void Main(string[] args)
  {
    int number = 20;
    AddFive(number);
    Console.WriteLine(number);
    Console.ReadKey();
  }

  static void AddFive(int number)
  {
    number = number + 5;
  }
```

در کد مثال فوق، یک متغیر عددی به نام number از نوع integer تعریف کرده و مقدار ۲۰ را به آن می دهیم. سپس این متغیر را به تابع ()AddFive ارسال نموده سپس مقدار ۵ واحد را به آن اضافه می کند. اما آیا متغیر واقعی number تغییر می کند؟

جواب خیر است، مقدار ۵ به نسخه کپی متغیر number اضافه شده و فقط درون تابع مورد اشاره تغییر کرده است. زیرا ما یک نسخه کپی از متغیر number را ارسال کرده ایم، نه رفرنس به شی اصلی آن. این روش حالت پیش فرض در C# بوده و در بیشتر موارد مقصود ما هم همین است.

اما اگر بخواهید مقدار شی اصلی متغیر number را تغیر دهید، بایستی کلید واژه ref را همانند کد زیر به پارامتر اضافه نمایید :

```
static void Main(string[] args)
{
   int number = 20;
   AddFive(ref number);
   Console.WriteLine(number);
   Console.ReadKey();
}

static void AddFive(ref int number)
{
   number = number + 5;
}
```

همانطور که در کد فوق مشاهده می کنید، ما علاوه بر این که در هنگام تعریف تابع، واژه کلیدی ref را به کار برده ایم، در هنگام فراخوانی تابع هم استفاده شده است.

اگر برنامه فوق را اجرا کنید، خواهید دید که مقدار اصلی متغیر number تغییر کرده است، نه نسخه کپی آن.

آموزش تغییر دهنده out modifier :

تغییردهنده out modifier بسیار شبیه ref عمل می کند. هر دو آن ها باعث می شود تا نسخه اصلی و رفرنس متغیر به عنوان پارامتر ارسال شود و نه مقدار آن. اما دارای دو تفاوت عمده با هم هستند :

۱) متغیری که به ref modifier ارسال می شود، حتما بایستی قبل از فراخوانی تابع مقداردهی اولیه شده باشد. اما این مورد درباره out صورت نکرده و شما می توانید یک متغیر مقداردهی نشده را به out ارسال کنید.

۲) از طرف دیگر شما نمی توانید بدون این که به یک متغیر out مقدار بدهید، تابع مورد نظر خود را به پارامتر out فراخوانی کنید. درست است که می توانید یک متغیر مقداردهی اولیه نشده را به یک تابع به عنوان out Parameter ارسال کنید، اما برای استفاده از آن در بدنه تابع، بایستی مقدار جدیدی را برای آن تعریف کنید.

آموزش تغییر دهنده Params modifier :

تا اینجا، توابعی که تعریف کردیم، تعداد مشخصی پارامتر را در هنگام تعریف یا فراخوانی دریافت می کردند. اما مواردی وجود دارد که ممکن است شما بخواهید تابع مورد نظرتان تعداد متفاوتی پارامتر را در هر بار فراخوانی دریافت و ارسال کند. این کار با ارسال یک متغیر مثل آرایه به تابع همانند روش زیر امکان پذیر است :

```
static void GreetPersons(string[] names) { }
```

البته فراخوانی این نوع تابع ها کمی پیچیده تر از حالت معمولی است، به صورت زیر :

```
GreetPersons(new string[] { "John", "Jane", "Tarzan" });
```

روش فوق کاملا درست، اما می توان آن را به صورت زیر بهتر بنویسید. با استفاده از واژه کلیدی Params :

```
static void GreetPersons(params string[] names) { }
```

کد فراخوانی فوق همانند فراخوانی به صورت زیر است :

```
GreetPersons("John", "Jane", "Tarzan");
```

مزیت دیگر استفاده از کلید واژه Params این است که شما می توانید پارامترهای خالی ((2re, Parameters را هم به تابع ارسال کنید.

در هنگام استفاده از Params، تابع می تواند انواع دیگری از پارامترها را نیز دریافت کند، منتهی پارامترهای Params بایستی در انتها تعریف شوند. از طرف دیگر فقط از یک پارامتر Params در تعریف و فراخوانی هر تابع می توان استفاده کرد.

کد مثال زیر، مثال کاملی از نحوه استفاده از Params در C# است :

```
static void Main(string[] args)
{
   GreetPersons(0);
   GreetPersons(25, "John", "Jane", "Tarzan");
   Console.ReadKey();
}

static void GreetPersons(int someUnusedParameter, params string[] names)
{
   foreach (string name in names)
      Console.WriteLine("Hello, " + name);
}
```

درس ۱۲ : آموزش کار با آرایه ها Arrays در C#

آموزش کار با آرایه ها Arrays در زبان C# :

آرایه ها Arrays مجموعه ای از آیتم ها مثل متن String می باشند. شما می توانید از آرایه ها برای قرار دادن چندین متغیر همسان در یک گروه و سپس انجام اعمال خاصی بر روی آن ها مثل مرتب سازی یا sorting استفاده کنید.

آرایه ها در C# تقریبا شبیه متغیرها تعریف می شوند با این فرق که یک [ ] در مقابل نوع داده ای آرایه قرار می گیرد. ساختار کلی تعریف یک آرایه در C# به صورت زیر است :

```
string[] names;
```

برای استفاده از یک آرایه نیاز دارید آن را تعریف اولیه و مقداردهی کنید. به صورت زیر :

```
string[] names = new string[2];
```

در کد فوق، عدد ۲ سایز آرایه را تعیین می کند. سایز آرایه تعداد اعضایی که می توانید در آرایه قرار دهید را مشخص می سازد. قرار دادن آیتم ها در یک Arrays کار ساده ای است. به صورت زیر :

```
names[0] = "John Doe";
```

اما چرا در تعریف اولین عضو آرایه از عدد ۰ استفاده کردیم. به دلیل این که در C# همانند سایر زبان های برنامه نویسی، شمارش واحدها به جای ۱ از ۰ شروع می شود. بنابراین اولین عضو آرایه با ۰ اندیس گذاری شده، دومی با ۱ و به این ترتیب.

شما بایستی به تعداد اعضای یک آرایه دقت کنید، زیرا تعریف عضو بیشتر از تعداد تعیین شده برای آرایه، برنامه را دچار خطا می کند. دقت کنید وقتی یک آرایه با ۲ عضو تعیین می شود، اعضای آن دارای اندیس های ۰ و ۱ هستند و عضوی با ۲ اندیس وجود ندارد. این یک اشتباه رایج در هنگام استفاده از آرایه هاست.

در درس های قبل تر، با ساختارهای تکرار (حلقه) در C# آشنا شدید، این ساختارها برای کار با آرایه ها بسیار مناسب هستند.

رایج ترین راه برای استخراج اطلاعات یک آرایه Arrays استفاده از حلقه ها یا loops می باشد. در هر بار تکرار حلقه، یک عضو آرایه استخراج شده و می توان عملیات مورد نظر خود را بر روی آن انجام داد. در مثال عملی زیر، نحوه خواندن و کار با یک آرایه را به وسیله حلقه loop نشان داده ایم :

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      string[] names = new string[2];

      names[0] = "John Doe";
      names[1] = "Jane Doe";

      foreach(string s in names)
        Console.WriteLine(s);

      Console.ReadLine();
    }
```

```
    }
}
```

در مثال فوق، از حلقه foreach استفاده کردیم که ساده ترین نوع حلقه برای کار با آرایه هاست. اما می توان از سایر انواع حلقه ها در C# نیز استفاده نمود، مثل حلقه for که به راحتی به وسیله آن می توان اعضای یک آرایه را شمرد و به تعداد اعضا پیام در خروجی چاپ کرد :

```
for(int i = 0; i < names.Length; i++)
    Console.WriteLine("Item number " + i + ": " + names[i]);
```

کارکرد کد فوق بسیار ساده است. ما از اندازه طول آرایه (Arrays Length) برای تعیین این که حلقه بایستی چند بار تکرار شود، استفاده کرده ایم. همچنین از شمارنده Counter(i) برای فهمیدن این که هر لحظه در کجای پردازش آرایه بوده و برای بیرون کشیدن هریک از اعضای آن استفاده نموده ایم. همانطور که در هنگام تعریف و مقداردهی آرایه از یک عدد به نام اندیس یا indexer استفاده کردیم، از همان عدد می توان برای خواندن و بیرون کشیدن اعضای آرایه استفاده نمود.

در بخش قبل گفتیم که می توان اعضای یک آرایه را مرتب یا sort کرد. این کار بسیار راحت است. کلاس Array Class شامل چندین متد (method) مختلف است که از آن ها می توانید برای کار با آرایه ها استفاده کنید. در مثال زیر اعداد به جای string یا متن استفاده کرده ایم تا منظور خاصی را نشان دهیم، وگرنه به همین سادگی می توان اعضای یک آرایه را از نوع string هم تعریف کرد. روش بسیار ساده تر دیگری نیز برای پر کردن و مقداردهی اعضای یک آرایه وجود دارد، به خصوص زمانی که اعضای آرایه شما مشخص و مرتب هستند. به صورت زیر :

```
int[] numbers = new int[5] { 4, 3, 8, 0, 5 };
```

فقط با استفاده از یک خط کد، آرایه ای با ۵ عضو را ایجاد نموده و پنج عدد یا integer را به عنوان اعضای آن مقداردهی کردیم.

با پر کردن اعضای یک آرایه به روش فوق، شما یک مزیت دیگر در کد خود خواهید داشت. با روش فوق، کامپایلر تعداد اعضای تعریف شده برای آرایه را با تعداد آیتم های شما چک کرده و اگر بیشتر از تعداد اعضا، آیتم ارائه دهید، خطا رخ می دهد. درواقع کد فوق را به صورت زیر می توان خلاصه تر نوشت، ولی در این حالت چک کردن خودکار کامپایلر را از دست می دهیم :

```
int[] numbers = { 4, 3, 8, 0, 5 };
```

اما بیایید نحوه مرتب کردن یا sort یک آرایه را باهم بررسی کنیم. مثال زیر را به دقت مطالعه کنید :

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] numbers = { 4, 3, 8, 0, 5 };

      Array.Sort(numbers);

      foreach(int i in numbers)
```

```
            Console.WriteLine(i);


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

تنها چیز جدید در کد مثال فوق، دستور Array.Sort است. این متد می تواند پارامترهای مختلفی را به عنوان ورودی دریافت کرده و در هر کدام یک نوع آرایه را مرتب خواهد کرد. مثلا نزولی، صعودی و یا . . . .

کلاس Array Class دارای متدهای مختلف دیگری برای کار با آرایه ها مثل متد ()Reverse است که به وسیله آن می توان ترتیب اعضای یک آرایه را معکوس کرد. برای درک بهتر، به لیست کامل متدهای آرایه ها در C# بروید.

آرایه هایی که در مثال های این درس استفاده کردیم، همه تک بعدی هستند، ولی آرایه ها در C# می توانند ۲ یا ۳ بعدی نیز باشند. به این آرایه ها آرایه های تو در تو یا ماتریس هم می گویند.

آرایه های چند بعدی در C# به ۲ دسته تقسیم می شوند :

- آرایه های مستطیلی یا Rectangular Array.
- آرایه های نامنظم یا Jagged Array.

فرق بین دو نوع آرایه فوق در این است که هر بعد آرایه های مستطیلی بایستی یک اندازه باشند، مثلا یک آرایه ها ۴×۴.

اما هر بعد آرایه های Jagged Array می توانند دارای سایزهای مختلفی باشند. بحث در مورد آرایه چند بعدی بسیار گسترده بوده و خارج از حوصله این آموزش می باشد.

درس ۱۳ : آموزش کار با کلاس ها در C#

آموزش کار با کلاس ها در C# :

در این درس قصد داریم تا شما را با مفهوم کلاس Class در C# و کاربرد آن ها آشنا کنیم. همچنین به بیان نحوه تعریف کلاس ها در برنامه و تعیین خواص و متدها برای آن ها خواهیم پرداخت.

اول از همه با مفهوم کلاس Class در C# شروع می کنیم. یک کلاس، مجموعه ای از خواص ها، متغیرها و متدهای مرتبط با هم است. یک کلاس خصوصیات ذکر شده را توصیف کرده و برای استفاده از آن در کد برنامه، یک نسخه از کلاس می سازید که به آن شی یا object می گویند. بر روی شی یا object ایجاد شده، می توانید متغیرها و متدهای کلاس را به کار ببرید. هر تعداد که نیاز داشته باشید می توانید شی یا object از روی کلاس ساخته و در نقاط مختلف کد برنامه استفاده کنید.

مبحث شی گرایی یا object oriented یک مقوله بسیار گسترده است که در این درس، درس های دیگر این بخش به مهم ترین جزئیات آن خواهیم پرداخت.

در بخش مقدمه آموزش C# و در مثال Hello World مشاهده کردید که از یک کلاس در کد برنامه استفاده شده بود در C# تقریبا همه چیز بر مبنای کلاس ها ایجاد می شوند. در کد این درس قصد داریم تا کلاس خود را گسترش داده و با انواع امکانات آن آشنا شویم :

```
using System;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      Car car;
```

```
        car = new Car("Red");
        Console.WriteLine(car.Describe());

        car = new Car("Green");
        Console.WriteLine(car.Describe());

        Console.ReadLine();
    }
}

class Car
{
    private string color;

    public Car(string color)
    {
        this.color = color;
    }

    public string Describe()
    {
        return "This car is " + Color;
```

```
    }

    public string Color
    {
      get { return color; }
      set { color = value; }
    }
  }
}
```

همانطور که مشاهده می کنید، در کد فوق یک کلاس جدید به نام Cor ایجاد کرده ایم. برای سهولت کار، این کلاس جدید را به همراه کلاس Program در فایل اصلی برنامه قرار داده ایم، اما روش رایج این است که هر کلاس درون فایل خود ایجاد شود. کلاس Car دارای یک متغیر به نام Color است که برای تعیین رنگ خودرو به کار می رود. خاصیت Color به صورت خصوصی یا Private تعیین شده و به این معناست که توابع و کلاس های خارج از تابع Car امکان دسترسی مستقیم به این خاصیت را ندارند. فقط توابع و متدهای کلاس Car می توانند به خاصیت Calor دسترسی مستقیم داشته باشند.

نکته : اگر تابع یا کلاسی خارج از کلاس Car بخواهد به خاصیت خصوصی یا Private به نام Color دسترسی داشته باشد، بایستی از یک Property برای این منظور استفاده کند. یک Property به نام Color در انتهای کد کلاس Car تعریف شده که وظیفه خواندن و نوشتن این خاصیت را بر عهده داشته و به سایر کلاس ها و توابع برنامه اجازه دسترسی به آن را می دهد.

همچنین کلاس Car دارای یک تابع سازنده یا Constructor می باشد. تابع سازنده، متدی است که به محض ساخته نشدن یک شیء از کلاس، اجرا می شود. تابع سازنده کلاس Car دارای یک پارامتر بوده که برای مقداردهی اولیه شی Car با خاصیت Color بزرگ قرمز (Red) به کار می رود. بنابراین هر شی از کلاس Car که ساخته شود، رنگ قرمز را به صورت پیش فرض برای خاصیت Color خود خواهد داشت. برای نشان دادن درستی این

مسئله نیز تابع Describe طراحی شده که نام هر شی یا object ساخته شده با مقدار رنگ آن را در خروجی نشان می دهد.

در قسمت های بعدی به آموزش مفاهیمی همچون خواص (Properties)، تابع سازنده (Constructor) و میدان دید (Visibility) خواهیم پرداخت.

درس ۱۴ : آموزش خواص Properties در کلاس C#

آموزش خواص Properties در کلاس های زبان C# :

خواص (Properties) در C# به شما امکان کنترل نحوه دسترسی و تغییر متغیرهای یک کلاس Class را می دهند. استفاده از خواص در C#، راه توصیه شده و درست جهت دسترسی به متغیرهای یک کلاس، به وسیله متدها یا توابع خارج از کلاس جاری، در زبان برنامه نویسی شی گرا object oriented می باشد. به عبارت دیگر، وقتی که یک متد یا کلاس دیگری در برنامه، بخواهد به متغیر یک کلاس دسترسی داشته باشد، بهتر است به جای این که مستقیما متغیر را استفاده کند، از طریق یک خاصیت یا Property در کلاس خود متغیر، به آن دسترسی داشته باشد.

در مثال آموزشی درس قبل (کار با کلاس ها) برای اولین بار نحوه استفاده از یک خاصیت یا Property را نشان دادیم. یک خاصیت یا Property همانند ترکیبی از یک متغیر Variable و یک متد method است. خاصیت نمی تواند پارامتری را دریافت کند، اما به وسیله آن می توانید قبل از ارسال و دریافت متغیر، مقدار آن را تغییر دهید. هر خاصیت یا Property از دو قسمت اصلی get و set تشکیل شده که همانند کد مثال زیر درون مجموعه Property تعریف می شوند :

```
private string color;


public string Color
{
   get { return color; }
   set { color = value; }
```

```
}
```

متد get مقدار متغیر را خوانده و آن را به برنامه بر می گرداند، از طرف دیگر متد set هم مقدار مورد نظر را به متغیر نسبت می دهد. مثال اول، ساده ترین حالت تعریف Property در C# است، اما می توانید آن را گسترش نیز بدهید.

نکته مهم : مسئله مهم در هنگام تعریف یک خاصیت یا Property این است که تعریف یکی از متدهای set یا get نیز کفایت می کند و در صورت تعریف یکی، تعریف متد دیگر اختیاری است. این مسئله، امکان تعریف خواص فقط خواندنی read-only و یا فقط نوشتنی write-only را می دهد.

در کد زیر یک مثال کامل تر از نحوه تعریف خاصیت ها در C# را نشان داده ایم :

```
public string Color
  {
    get
    {
      return color.ToUpper();
    }
    set
    {
      if (value == "Red")
        color = value;
      else
        Console.WriteLine("This car can only be red!");
    }
  }
```

در کد مثال فوق، خاصیت را کمی گسترش دادیم. در کد جدید، متغیر Color در هنگام return، به دلیل استفاده از متد ()ToUpper به صورت حروف بزرگ، برگردانده می شود. از طرف دیگر، با استفاده از دستور شرطی if، فقط مقدار رنگ "red" برای متغیر Color پذیرفته می شود.

درس ۱۵ : آموزش Constructor و destructor در زبان C#

آموزش کار با تابع سازنده Constructor در زبان C# :

تابع سازنده یا Constructor متد ویژه ای است که هر بار به محض ساخته شدن یک شی یا object از کلاس، اجرا می شود. درواقع از تابع سازنده برای مقداردهی اولیه متغیرها یا اجرای یک کد ثابت استفاده می شود.

یک تابع سازنده هیچ گاه مقدار خروجی یا return ندارد و به همین دلیل در تعریف آن، هیچ نوع متغیری جهت return تعریف نمی شود. ساختار کلی تعریف تابع سازنده Constructor در کلاس های C# به صورت زیر است :

```
public string Describe()
```

همچنین تابع سازنده را به صورت زیر نیز می توانید تعریف کنید :

```
public Car()
```

در مثال عملی این درس، کلاسی به نام Car Class داریم که دارای یک تابع سازنده بوده که یک پارامتر متنی String را به عنوان ورودی دریافت می کند. البته تابع های سازنده در C# می توانند overload نیز شوند. یعنی این که ما چندین تابع سازنده هم نام برای یک کلاس داشته باشیم، ولی پارامترهای ورودی آن ها با هم متفاوت باشد. کلاس زیر یک کد عمل را نشان می دهد :

```
public Car()
{
```

```
}

public Car(string color)
{
  this.color = color;
}
```

یک تابع سازنده، می تواند تابع سازنده دیگر را فراخوانی کند که به صورت های مختلف ممکن است کد زیر یک مثال در این زمینه ارائه داده است :

```
public Car()
{
  Console.WriteLine("Constructor with no parameters called!");
}

public Car(string color) : this()
{
  this.color = color;
  Console.WriteLine("Constructor with color parameter called!");
}
```

اگر متد مثال فوق را اجرا کنید، خواهید دید که تابع سازنده ای که هیچ پارامتری را به عنوان ورودی ندارد، ابتدا اجرا می شود. از این حالت برای مقداردهی اشیا (objects) های یک کلاس با یک تابع سازنده پیش فرض استفاده می شود.

اگر بخواهید که تابع سازنده دار ای پارامتر فراخوانی شود، می توانید یه صورت کد زیر عمل کنید :

```
public Car(string color) : this()
```

```
{
  this.color = color;
  Console.WriteLine("Constructor with color parameter called!");
}

public Car(string param1, string param2) : this(param1)
{

}
```

نکته : اگر شما تابع سازنده ای که دارای ۲ پارامتر است را فراخوانی کنید، پارامتر اول برای فراخوانی تابع سازنده ای که دارای ۱ پارامتر است، استفاده می شود.

آموزش کار با تابع تخریب کننده یا Destractor در C# :

تابع تخریب کننده یا Destractor در زبان C#، متدی است که در هنگام از بین رفتن یک شی از کلاس، اجرا می شود. زبان C#، یک زبان پاتک کننده خودکار سیستم یا garbage collector است، به این معنی که object هایی که دیگر در برنامه نیاز ندارید را جهت خالی کردن حافظه و آزاد نمودن سیستم، پاک می کند.

از طرف دیگر در برخی موارد شاید نیاز داشته باشید تا یک Clean up در سیستم انجام دهید، اینجاست که تابع های تخریب کننده Destructor به کار می آیند.

تابع های تخریب کننده چندان شبیه سایر متدها در زبان C# نیستند. در کد عملی زیر یک مثال از تابع تخریب کننده نشان داده شده است :

```
~Car()
{
  Console.WriteLine("Out..");
}
```

به محض این که شی یا object ایجاد شده از کلاس، توسط تمیز کننده خودکار garbage collector جمع آوری شده، متد فوق فراخوانی می شود.

درس ۱۶ : آموزش کار با Method overloading در C#

آموزش کار با روش Method overloading در C# :

بسیاری از زبان های برنامه نویسی از تکنیک ای به نام پارامترهای پیش فرض /اختیاری (defult/optional parameters) پشتیبانی می کنند. این تکنیک به برنامه امکان می دهد تا با تعیین مقدار پیش فرض برای یک یا چند پارامتر تابع، آن ها را در هنگام مقدار دهی اختیار کند. این روش برای افزودن انعطاف پذیری به کد برنامه، بسیار کاربرد دارد.

برای مثال، می خواهید قابلیت کارکرد را به تابع ای که یک یا چند پارامتر ورودی دارد، بدهید. در اینگونه موارد، به دلیل عدم ارسال تعداد مورد نیاز پارامتر در هنگام فراخوانی، ممکن است کد شما درست اجرا نشود. برای حل این مسئله، می توانید از امکان جدید تعیین پارامترهای اختیاری یا optional استفاده کنید. در این روش، شما برای برخی پارامترها یک مقدار پیش فرضی یا default تعیین کرده که حتی اگر در هنگام فراخوانی تابع، مقداری برای آن ارسال نشد، کد دچار مشکل نشود.

پارامترهای پیش فرض (default parameters) در زبان C# 4.0 معرفی شدند، اما تا قبل از آن برنامه نویسان از تکنیکی تقریبا مشابه به نام method overloading استفاده می کردند. در این حالت، برنامه نویس چندین تابع هم نام ولی با مجموعه پارامترهای مختلف را تعریف می کند. برای مثال متد اول یک پارامتر و متد دوم دو پارامتر دریافت می کند. یک مثال مناسب برای این روش، تابع Substring از کلاس String Class است. به صورت زیر :

```
string Substring (int startIndex)
string Substring (int startIndex, int length)
```

شما می توانید تابع فوق را با یک یا دو پارامتر فراخوانی کنید. اگر تابع را با یک پارامتر فراخوانی کنید حالت اول و اگر با دو پارامتر فراخوانی کنید، حالت دوم اجرا می شود.

بنابراین با تعیین شکل های مختلف از یک تابع، می توانید حجم کد نویسی را تا حدود زیادی کاهش دهیم. برای این منظور، کاری می کنیم تا متد ساده و معمولی، کد سایر متدها را تولید و اجرا کند. به مثال زیر دقت کنید :

```
class SillyMath
{
  public static int Plus(int number1, int number2)
  {
    return Plus(number1, number2, 0);
  }

  public static int Plus(int number1, int number2, int number3)
  {
    return number1 + number2 + number3;
  }
}
```

در کد مثال فوق، تابع Plus را با دو حالت تعریف کرده ایم. در حالت اول، تابع دو پارامتر را جهت جمع کردن ۲ عدد دریافت می کند، درحالی که حالت دوم سه پارامتر دارد. درواقع کار اصلی را نسخه ۳ پارامتری تابع انجام می دهد. اگر بخواهیم دو عدد را جمع کنیم، خیلی ساده، تابع با حالت ۳ پارامتری را فراخوانی کرده و عدد ۰ را به پارامتر سوم پاس می دهیم. تا به عنوان مقدار پیش فرض برای آن استفاده شود. کد مثال فوق، منظور ما در بخش قبل را نشان می دهد.

حال اگر بخواهید تا ۴ عدد را با هم جمع کنید، می توانید یک نسخه ۴ پارامتری دیگر را نیز به برنامه اضافه کنید. به صورت کد زیر :

```
class SillyMath
{
  public static int Plus(int number1, int number2)
  {
    return Plus(number1, number2, 0);
  }

  public static int Plus(int number1, int number2, int number3)
  {
    return Plus(number1, number2, number3, 0);
  }

  public static int Plus(int number1, int number2, int number3, int number4)
  {
    return number1 + number2 + number3 + number4;
  }
}
```

مثال فوق خیلی ساده تکنیک method overloading را در C# نشان داده و نحوه ارتباط دادن توابع با هم را بیان می کند.

درس ۱۷ : آموزش تعیین میدان دید Visibility در C#

آموزش تعیین میدان دید Visibility در C# :

میدان دید یک کلاس، متد، متغیر یا خاصیت در C#، نحوه امکان دسترسی به آن عنصر و دیده شدن در سطح کل برنامه را تعیین می کنید. به این امکان در اصطلاح سطح دسترسی یا Visibility می گویند.

رایج ترین حالت ها برای Visibility در C#، خصوصی Private و عمومی Public است، اما حالت های دیگری نیز وجود دارد که در لیست زیر به معرفی آن ها پرداخته ایم. برخی از موارد زیر ممکن است تاکنون مورد استفاده شما قرار گرفته باشد، اما دانستن آن ها ضروری است.

ـ عمومی Public : در این حالت عنصر به صورت عمومی تعریف شده و از هر جای برنامه توسط هر عنصر دیگر مثل سایر کلاس ها و توابع قابل دسترسی است. این حالت دارای حداقل محدودیت برای عنصر بوده و Enums و Interface ها به صورت پیش فرض public هستند.

ـ محافظت شده یا Protected : در این حالت عنصر فقط توسط عوامل هم کلاس خود یا عوامل موجود در کلاس هایی که از کلاس آن به ارث رفته اند، قابل دسترس است.

ـ درونی یا internal : در این حالت عنصر فقط درون پروژه جاری قابل دسترسی است.

ـ درونی محافظت شده یا Protected internal : این حالت، همانند حالت internal است با این تفاوت که عناصر موجود در کلاس هایی که از کلاس عنصر به ارث رفته اند، حتی اگر در پروژه های دیگر باشند قابلیت دسترسی به آن را دارند.

ـ خصوصی یا Private : در این حالت فقط عوامل هم کلاس عنصر امکان دیدن و دسترسی به آیتم مورد نظر را دارند. این حالت دارای بیشترین میزان محدودیت بوده و Class ها و Struct ها به صورت پیش فرض خصوصی private هستند.

برای مثال، اگر شما دارای دو کلاس به نام های Class 1 و Class 2 باشید، اعضای private مربوط به Class 1 فقط درون خود آن کلاس قابل دسترس هستند. شما نمی توانید یک نمونه از اشیای Class 1 را در Class 2 ساخته و امکان دسترسی به آن ها را داشته باشید.

اما اگر Class 2 از Class 1 به ارث رفته یاشد (فرزند آن باشد) که در اصطلاح می گوییم inherit شده، فقط اعضای غیر private کلاس ۱ در کلاس ۲ قابل دسترس هستند.

درس ۱۸ : آموزش مفهوم Static members در کلاس های C#

آموزش مفهوم Static members در کلاس های C# :

همان طور که در درس های قبل متوجه شدید، رایج ترین راه تعامل و ارتباط با یک کلاس Class، ساخت یک نمونه از آن (شی یا object) و سپس کار کردن بر روی object مورد نظر است. در بیش تر موارد، این کل چیزی است که کلاس شامل می شود، یعنی ساخت چندین نمونه مختلف از یک کلاس و سپس استفاده از هر کدام از آن ها در یک قسمت و یا یک منظور. اما مواردی وجود دارد که تمایل دارید کلاسی داشته باشید که بدون نمونه سازی از آن یا حداقل بدون نیاز به ساخت شی از آن، بتوانید از اعضا و متدهای آن استفاده کنید. برای مثال ممکن است کلاسی داشته باشید که دارای یک متغیر بوده و مقدار این متغیر در همه موارد صرف نظر از محل به کارگیری آن، یکسان است. به این موارد اعضای ثابت یا Static members می گویند، ثابت Static به این دلیل که همواره مقدار آن ها در سطح برنامه یکسان است.

یک کلاس می تواند استاتیک Static بوده و یا دارای اعضا و متدهای Static باشد. یک کلاس ثابت Static Class را نمی توانید نمونه سازی کنید. کلاس های ثابت در واقع مجموعه ای member ها هستند تا یک کلاس به معنای واقعی.

همچنین شما ممکن است یک کلاس معمولی (not-static) ساخته و برای اعضای آن، ثابت تعریف کنید. یک کلاس غیر ثابت not-static را می توانید نمونه سازی کرده و از روی آن object بسازید، اما نمی توانید از اعضای ثابت آن در شی ها استفاده کنید.

کد زیر یک مثال از نحوه تعریف کلاس های ثابت Static Class می باشد :

```
public static class Rectangle
{
  public static int CalculateArea(int width, int height)
  {
    return width * height;
  }
```

```
}
```

همانطور که مشاهده کردید، ما از کلمه کلیدی Static برای مشخص کردن کلاس از نوع ثابت استفاده کرده و همچنین این کلمه کلیدی را نیز برای متد Calculator Area به کار برده ایم. اگر هم برای کلاس و هم برای متد، کلمه کلیدی Static را استفاده نکنید، کامپایلر C# دچار خطا می شود. زیرا یک عضو غیر ثابت (not-Static) را نمی توان در یک کلاس Static تعریف کرد.

برای استفاده از متد Calculator، بایستی آن را همانند کد زیر و به صورت مستقیم با کلاس فراخوانی کرد :

```
Console.WriteLine("The area is: " + Rectangle.CalculateArea(5, 4));
```

ما می توانیم متدهای کاربردی دیگری را نیز درون کلاس Rectangle Class تعریف کنیم اما شاید این سوال برایتان پیش آمده، که چرا ما پارامترهای width و height را به جای تعریف در کلاس، مستقیما به متد مقصد پاس داده ایم؟

به دلیل این که کلاس فوق به صورت ثابت Static تعریف شده، ما می توانیم مقدار متغیرهای height و width را درون کلاس نگهداری کنیم، اما به صورت ثابت و فقط دارای یک نمونه از مقادیر مثلا ۳ و ۲ و دیگر نمی توان آن ها را تغییر داد. این نکته مهم را بایستی در کار با توابع Static لحاظ کنید.

به جای کد فوق، می توان با غیر ثابت تعریف کردن کلاس، امکان تغییر و انعطاف پذیری کامل را به کلاس خود بدهید. همانند کد مثال زیر :

```
public class Rectangle
{
  private int width, height;

  public Rectangle(int width, int height)
  {
```

```
        this.width = width;
        this.height = height;
    }

    public void OutputArea()
    {
        Console.WriteLine("Area output: " + Rectangle.CalculateArea(this.width, this.height));
    }

    public static int CalculateArea(int width, int height)
    {
        return width * height;
    }
}
```

همانطور که مشاهده می کنید، کلاس را به صورت غیر ثابت non-Static تعریف کرده ایم. همچنین یک تابع سازنده Construction تعریف شده که مقدار پارامترهای Width و height را به نمونه ساخته از کلاس، نسبت می دهد. در پایان هم متد out put Area را جهت محاسبه مساخت به صورت static تعریف کرده ایم. این مثال نمونه خوبی از ترکیب اعضای غیر ثابت و ثابت در یک کلاس non-static می باشد.

درس ۱۹ : آموزش مفهوم ارث بری Inhertance در کلاس C#

آموزش مفهوم ارث بری Inhertance در کلاس های C# :

یکی از جنبه های کلیدی زبان های برنامه نویسی شی گرا Objected Oriented Programming (OOP) از جمله C# بر پایه آن بنا شده است، مفهوم ارث بری یا Inhertance می باشد. ارث بری یعنی تولید کلاس هایی جدید که برخی از ویژگی های خود را از کلاس مادر Parent Class به ارث برده اند.

کل چهارچوب کاری NET، بر پایه مفهوم ارث بری نباشد، که جمله معروف "همه چیز شی یا object است"، از جمله نتایج آن می باشد. در زبان C#، حتی یک عدد ساده، نمونه ای از یک کلاس است که خود از کلاس مادر System.object به ارث رفته است. اگرچه چهارچوب کاری NET.، این امکان را نیز برای شما فراهم کرده تا مقدار مورد نظر خود را مستقیما به یک عدد بدهید، بدون آن که نیاز داشته باشید تا نسخه جدیدی از کلاس Integer ایجاد کنید.

شاید بیان مفهوم ارث بری کمی سخت باشد، اما ارائه مثال های عملی کمک شایانی در این زمینه می کند. نمونه کد زیر، یک مثال ساده از مفهوم ارث بری در زبان C# است :

```
public class Animal
{
  public void Greet()
  {
    Console.WriteLine("Hello, I'm some sort of animal!");
  }
}

public class Dog : Animal
{


}
```

در کد فوق، در مرحله اول یک کلاس به نام Animal Class تعریف کرده ایم و دارای یک متد ساده جهت چاپ پیام خوش آمد گویی است. سپس کلاس Dog Class را ایجاد کرده و با قرار دادن یک : در مقابل آن، به C# اعلام کرده ایم این کلاس بایستی از کلاس Animal به ارث رود. نکته جالب در مورد مثال فوق این است که رابطه سگ و حیوان نیز در طبیعت به همین صورت می باشد، یعنی Dog زیرمجموعه ای از حیوانات است. اکنون بیایید نحوه به کار بردن کلاس ها را بررسی کنیم :

```
Animal animal = new Animal();
animal.Greet();
Dog dog = new Dog();
dog.Greet();
```

اگر کد مثال فوق را اجرا نمایید، خواهید دید علی رغم این که متد ()Greeting را برای کلاس Dog تعریف نکرده ایم، اما این کلاس می تواند به پیام پیام خوش آمد بگوید، زیرا متد ()Greeting را از کلاس مادر خود یعنی Animal Class به ارث برده است.

البته پیام خوش آمدگویی اولیه کد فوق، کمی نامفهوم است. بیایید کدی به مثال قبلی اضافه کنیم تا بدانیم چه حیوانی به ما خوش آمد می گوید!

```
Animal animal = new Animal();
animal.Greet();
Dog dog = new Dog();
dog.Greet();
```

در کنار متد اضافه شده به کلاس Dog Class، دو چیز مهم را مشاهده می کنید. ما کلمه کلیدی مجازی (Virtual keyword) را به متد کلاس Animal Class اضافه کرده ایم، از طرف دیگر کلمه کلیدی averide را نیز برای کلاس Dog Class به کار برده ایم.

در زبان C#، شما نمی توانید عضو (member) یک کلاس را بازنویسی یا averide کنید، مگر این که آن را به عنوان Virtual تعیین کرده باشید. اگر هم بخواهید، می توانید متد به ارث برده شده را با به کار بردن کلمه کلیدی base، به صورت اولیه خود فراخوانی کنید. به صورت کد مثال زیر :

```
public override void Greet()
```

```
{
  base.Greet();
  Console.WriteLine("Yes I am - a dog!");
}
```

تنها متدها (Methods) نیستند که می توان آن ها را در کلاس ها به ارث برد، سایر خصوصیات یک کلاس مثل فیلدها (Fileds) و خواص (Properties) نیز خاصیت ارث بری دارند. فقط بایستی به قواعد میدان دید Visibility که در درس قبل تشریح کردیم، توجه کنید.

ارث بری یا Inhertance فقط مختص یک کلاس به کلاس دیگر نیست. شما می توانید یک سلسله مراتب از کلاس های مرتب به هم را بسازید. برای مثال می توانید یک کلاس به نام Puppy ایجاد کرده که خودش از کلاس Dog به ارث برود، در حالی که Dog فرزند کلاس Animal است.

نکته مهم : شما در C# نمی توانید کاری کنید که یک کلاس از چند کلاس مختلف به ارث برود. به این کار ارث بری چندگانه (multiple inheritance) می گویند که در C# پشتیبانی نمی شود.

درس ۲۰ : آموزش مفهوم کلاس پایه Abstract Class در C#

آموزش مفهوم کلاس پایه Abstract Class در زبان C# :

کلاس های مطلق یا پایه (Abstract Class) در زبان C#، که با کلمه کلیدی abstract مشخص می شوند، کلاس های پایه و مادر در یک سلسله مراتب درختی کلاس ها می باشند. به عبارت دیگر این کلاس ها، کلاس مرجع بوده و بقیه کلاس ها به ترتیب از روی این کلاس به ارث می روند.

نکته : مسئله مهم در مورد کلاس های ثابت Abstract Class این است که شما نمی توانید یک نمونه یا شی از روی این کلاس ها بسازید، اگر این کار را انجام دهید، با خطای کامپایلر مواجه خواهید شد.

به جای عدم امکان ساخت شی از روی کلاس های پایه، می توانید یک کلاس فرزند یا Subclass همانطور که در درس قبل آموزش دادیم، را از روی کلاس پایه ساخته و سپس object های خود را از روی کلاس فرزند تولید کنید.

اما چه زمانی ممکن است به کلاس های پایه نیاز داشته باشید؟ اگر بخواهیم با هم رو راست باشیم، شما ممکن است یک پروژه کامل را کدنویسی کنید، بدون این که نیازی به کلاس پایه داشته باشید. اما این نوع کلاس ها برای یک منظور خاص، بسیار مناسب بوده و آن کاربرد در چهارچوب کاری یا FrameWork ها می باشد. به همین دلیل است که چهارچوب کاری .NET پر از کلاس های پایه است.

در کد کلاس زیر، ما یک کلاس پایه به نام Four Legged Animal (چهارپایان) را ایجاد کرده و سپس کلاس دیگری به نام Dog تعریف نموده که از کلاس پایه به ارث می رود.

```
namespace AbstractClasses
{
  class Program
  {
    static void Main(string[] args)
    {
      Dog dog = new Dog();
      Console.WriteLine(dog.Describe());
      Console.ReadKey();
    }
  }

  abstract class FourLeggedAnimal
  {
    public virtual string Describe()
```

```
    {
        return "Not much is known about this four legged animal!";
    }
  }

  class Dog : FourLeggedAnimal
  {

  }
}
```

اگر کد مثال فوق را با مثال های درس قبل، آموزش مفهوم ارث بری در C# مقایسه کنید، تفاوت چندانی بین آن ها مشاهده نخواهید کرد. درواقع کلمه کلیدی abstract در جلوی نام کلاس FourLeggedAnimal مهم ترین تفاوت این مثال هاست. همانطور که مشاهده می کنید، ما یک نمونه از کلاس Dog را ایجاد کرده و سپس متد به ارث برده شده Describe() را از کلاس FourLeggedAnimal فراخوانی کرده ایم. اکنون بیایید یک نمونه از کلاس FourLeggedAnimal را ایجاد کنیم، به صورت کد زیر :

```
FourLeggedAnimal someAnimal = new FourLeggedAnimal();
```

اما شما با خطای کامپایلر مواجه خواهید شد. شرح خطا به صورت زیر است :

```
<i>Cannot create an instance of the abstract class or interface 'AbstractClasses.FourLeggedAnimal'</i>
```

همان طور که در مثال قبل دیدید، ما متد به ارث رفته Describe() را فراخوانی کردیم، اما در حقیقت کاربرد چندانی برای ما نداشت. لذا بیایید توسط کد زیر آن را بازنویسی یا override کنیم :

```
class Dog : FourLeggedAnimal
```

```
{
  public override string Describe()
  {
    return "This four legged animal is a Dog!";
  }
}
```

در کد مثال فوق، به طور کامل متد را بازنویسی یا Override کردیم. اما گاهی ممکن است شما بخواهید رفتار را از کلاس پایه به ارث برده و مقدار کد نیز به آن اضافه کنید. برای این منظور بایستی از کلمه کلیدی base استفاده نموده که اشاره به کلاسی دارد که از آن inherit کرده ایم. به صورت کد زیر :

```
abstract class FourLeggedAnimal
{
  public virtual string Describe()
  {
    return "This animal has four legs.";
  }
}


class Dog : FourLeggedAnimal
{
  public override string Describe()
  {
    string result = base.Describe();
    result += " In fact, it's a dog!";
```

```
        return result;
    }
}
```

اکنون می توانید کلاس های فرزند subclass دیگری را نیز برای کلاس FourLeggedAnimal درست کنید.

در درس بعدی، یک مثال پیشرفته تر را ارائه داده و نحوه کار با متدهای پایه abstract methods)) را آموزش خواهیم داد.

درس ۲۱ : مطالعه کامل تر کلاس های پایه Abstract Class در C#

مطالعه کامل تر کلاس های پایه Abstract Class در C# :

در درس قبلی، با مفهوم کلاس پایه Abstract Class در زبان C# آشنا شدیم. در این درس، قصد داریم مثال درس قبل را کمی گسترش داده و در آن از متدهای ثابت Abstract Methods نیز استفاده کنیم. کد تعریف کلاس های ثابت همانند یک متد معمولی است ولی کدی درون آن ها نوشته نمی شود :

```
abstract class FourLeggedAnimal
{
    public abstract string Describe();
}
```

اما چرا می خواهیم یک متد خالی را تعریف کنیم، در حالی که کاری برایمان انجام نمی دهد؟

به دلیل این که متد پایه (Abstract Method) یک تعمد یا اجبار است تا آن متد در تمامی کلاس های فرزند کلاس جاری اجرا شود. درواقع، به وسیله این کار در زمان اجرای برنامه چک می کنیم، آیا تمامی کلاس های فرزند، این تابع را در خود تعریف کرده اند یا خیر.

یک بار دیگر اشاره داریم که این روش بهترین روش برای تعیین یک کلاس پایه برای چیزی در برنامه است، در حالی که می توانیم کنترل کنیم، کلاس فرزند یا subclass چه کارهایی را بایستی قادر باشد، انجام دهد.

با در نظر گرفتن مطلب فوق، شما همواره می توانید کاری کنید تا یک کلاس فرزند subclass همانند کلاس پایه عمل کند. برای مثال، به کد مثال زیر دقت کنید :

```
namespace AbstractClasses
{
  class Program
  {
    static void Main(string[] args)
    {
      System.Collections.ArrayList animalList = new System.Collections.ArrayList();
      animalList.Add(new Dog());
      animalList.Add(new Cat());
      foreach(FourLeggedAnimal animal in animalList)
        Console.WriteLine(animal.Describe());
      Console.ReadKey();
    }
  }

  abstract class FourLeggedAnimal
  {
    public abstract string Describe();
  }


  class Dog : FourLeggedAnimal
  {
```

```
    public override string Describe()
    {
      return "I'm a dog!";
    }
  }

  class Cat : FourLeggedAnimal
  {
    public override string Describe()
    {
      return "I'm a cat!";
    }
  }
}
```

همانطور که در کد مثال مشاهده می کنید، ما یک لیست آرایه ای ArrayList شامل نام چند حیوان را درست کرده ایم. سپس یک نمونه جدید از Dog و Cat را ایجاد و به لیست اضافه کرده ایم. در شی ذکر شده، نمونه هایی ایجاد شده از کلاس Dog و Cat هستند، از طرف دیگر نوعی از کلاس FourLeggedAnimal نیز محسوب می شوند. تا زمانی که کامپایلر می داند که کلاس فرزند متعلق به کلاس پایه است، می توانید متد Describe() را فراخوانی و اجرا کنید، بدون این که دقیقا نوع حیوان شی را بدانید.

بنابراین با انجام عمل تبدیل نوع داده ای (TypeCasting) که در حلقه foreach loop برای کلاس FourLeggedAnimal انجام دادیم، به اعضای کلاس فرزند subclass دسترسی پیدا کردیم. این کار در بسیاری از موارد سودمند خواهد بود.

درس ۲۲ : آموزش کار با Interface ها در C#

آموزش کار با Interface ها در زبان C# :

در درس قبل، به بررسی کلاس های پایه یا Abstract ها در C# پرداختیم. Interface ها در C# نیز تا حد زیادی مشابه Abstract Class ها بوده و در این ویژگی که نمی توان از روی آن یک نسخه یا شی ساخت با هم یکسان هستند.

با این حال، Interface ها حتی از کلاس های Abstract نیز مفهومی تر هستند، به دلیل این که بدنه دستورات متد (method body) در آن ها اصلا مجاز نیست (یعنی نمی توان دستوراتی را برای متدهای تعریف شده تعیین کرد). از آنجایی که در Interface ها متد با کد واقعی وجود ندارد، بنابراین به فیلدها Fields نیز نیازی نیست. اما تعیین خواص Propertoes، اندیس ها indexer و رویدادها event امکان پذیر است.

شما می توانید یک Interface را به عنوان یک Contract (قرارداد) نیز در نظر بگیرید، یعنی کلاسی که اجرای آن مستلزم اجرای تمامی متدها methods و خواص Properties کلاس است.

نکته : نکته مهم درباره Interface ها این است که از آنجایی که C# امکان به ارث بری چندگانه (multiple inheritance) را نمی دهد، یعنی یک کلاس از بیش از یک کلاس پایه به ارث برود، اما اجازه ای Interface های چندگانه را می دهد.

اما تمامی این موارد گفته شده در کد، به چه شکل خواهد بود. کد مثال زیر یک نمونه خوب از Interface ها در C# است. به کد مثال دقت کنید، در ادامه توضیحاتی راجع به آن ارائه خواهیم داد :

بیایید از میانه کد، جایی که Interface را تعریف کرده ایم، شروع کنیم. همانطور که متوجه شدید، تنها تفاوت در تعریف یک Interface به جای Class واژه کلیدی به کار رفته است که interface به جای کلاس نوشته شده است. همچنین، قبل از نام Interface یک I را برای تلقی عبارت interface قرار داده که این فقط یک استاندارد کد نویسی بوده و نیازی به رعایت آن نیست. شما می توانید Interface مورد نظر خود را در هر جای برنامه که

بخواهید فراخوانی کنید، و از آنجا که فراخوانی آن ها شبیه فراخوانی کلاس ها است، قرار دادن I قبل از نام Interface به درک بهتر کد توسط کدنویس کمک می کند.

سپس متد Describe را تعریف کرده و بعد از آن خاصیت Name Property را نوشته که دارای هر دو واژه کلیدی get و set بوده و به این معناست که یک خاصیت هم نوشتنی و هم خواندنی است.

نکته مهم : همانطور که متوجه شدید، خواص تعیین کننده میدان دید یا سطح دسترسی (access modifiers) مثل public، private، protected و ... در ابتدای نام Interface قرار داده نشده است. زیرا این خواص در Interface ها مجاز نبوده و آن ها همه به صورت پیش فرض public هستند.

در مرحله بعدی کلاس Dog class را قرار داده ایم. سپس با قرا دادن یک : بین نام کلاس و Interface مورد نظر، کلاس Dog از Interface به ارث رفته است، دقیقا مثل کلاس ها. اما در کد فوق دو Interface را برای کلاس Dog به کار برده و آن ها را با کاما (،) از هم جدا کرده ایم. شما می توانید هر تعداد Interface که بخواهید را در مقابل نام کلاس تعریف کنید، اما در این مثال ما فقط دو Interface به نام های IAnimal و IComparable را به کلاس خود نسبت داده ایم. IComparable یک Interface مشترک بین کلاس هایی که قابلیت مرتب شدن (sorting) دارند، می باشد. نتیجه گیری این که، در کد فوق ما هم یک method و یک property را از IAnimal Interface در کد استفاده کرده و همزمان متد Compare To را از IComparable interface را به کار برده ایم.

اکنون ممکن است با خودتان فکر کنید، اگر همه کارها را ما خودمان باید انجام داده و کلیه متدها و خواص Properties لازم را در کد تعریف کنیم، چرا این قدر برای نوشتن Interface به خودمان زحمت بدهیم.

جواب سوال فوق را به خوبی در بخش بالایی کد مثال می توانید مشاهده کرده و متوجه شوید چرا این قدر زمان گذاشتن، با ارزش خواهد بود.

در کد مثال فوق، ما تعدادی Dog را به عنوان object های یک list تعریف نموده و سپس لیست خود را مرتب یا sort کرده ایم. اما list از کجا می داند که چگونه نام سگ ها را مرتب کند؟ به دلیل این که، کلاس Dog Class دارای متدی به نام CompareTo است که به آن می گوید چگونه دو Dog را با هم مقایسه کند.

اما چگونه list می کند که شی Dog object می تواند عمل مرتب سازی انجام داده و بایستی کدام متد را برای مقایسه سگ ها فراخوانی کند. به دلیل اینکه ما این کار را با تعیین یک Interface مناسب و نسبت دادن آن به کلاس، به list اعلام کرده ایم. Interface بایستی متد CompareTo را اجرا کرده و این زیبایی استفاده از Interface ها در کد است.

درس ۲۳ : آموزش اشکال زدایی Debugging در پروژه های C#

آموزش اشکال زدایی کدها (Debugging) در زبان C# :

در پروژه های بزرگ، معمولا حجم کدنویسی به اندازه ای زیاد می شود که گاهی اوقات برنامه نویسان خودشان هم کاملا بر روی کد پروژه تسلط نداشته و جز با اجرای آن نمی توانند موفقیت برنامه را در عمل متوجه بشوند.

```
class Program
  {
    static void Main(string[] args)
    {
      List<dog> dogs = new List<dog>();
      dogs.Add(new Dog("Fido"));
      dogs.Add(new Dog("Bob"));
      dogs.Add(new Dog("Adam"));
      dogs.Sort();
      foreach(Dog dog in dogs)
        Console.WriteLine(dog.Describe());
      Console.ReadKey();
    }
  }
```

```
interface IAnimal
{
  string Describe();

  string Name
  {
    get;
    set;
  }
}

class Dog : IAnimal, IComparable
{
  private string name;

  public Dog(string name)
  {
    this.Name = name;
  }

  public string Describe()
  {
    return "Hello, I'm a dog and my name is " + this.Name;
  }
```

```
    public int CompareTo(object obj)
    {
      if(obj is IAnimal)
        return this.Name.CompareTo((obj as IAnimal).Name);
      return 0;
    }

    public string Name
    {
      get { return name; }
      set { name = value; }
    }
  }
}
      </dog></dog>
```

چیزی که شما به آن نیاز دارید، تقریبا همانند کلاه شعبده بازهاست تا به وسیله آن بتوانید درون برنامه خود را باز کرده و در هنگام اجرا، ببندید واقعا چه پروسه ای در جریان است؟!

عمل اشکال زدایی کدها یا Debugging همان کلاه شعبده بازی مورد نظر است. ابزاری که پس از آشنا شدن و یادگیری کار با آن، تقریبا دیگر نمی توان بدون استفاده از Debugging کد بنویسید. Debugging ابزاری است که هر برنامه نویس بایستی آن را به خوبی شناخته و کار با آن را یاد بگیرید. زیرا عملا بدون استفاده از ابزار، اصلاح اشکالات یا bug های کدتان غیر ممکن است.

ساده ترین حالت اشکال زدایی یا Debugging، که هنوز حتی توسط کد نویسان حرفه ای مورد استفاده قرار می گیرد، روش "print debugging" است. روشی که در آن برنامه نوشته یا عددی را در جاهای مختلف کد نشان می دهد تا متوجه شوید که در هر زمان چه بخشی از کد در حال اجراست و هر متغیر در هر مرحله دارای چه مقداری است.

در زبان C#، می توانید از متد CONSOLE.write برای نمایش مقدار یک متغیر در خروجی و یا چاپ یک عبارت متنی استفاده کنید. این متن ها و مقادیر بر روی صفحه یا Console ظاهر می شوند. این کار در برخی موارد کافی به نظر می رسد. اما اگر شما یک نرم افزار IDE خوب مثل Visual Studio کار می کنید، ابزارهای بسیار کاربردی تر و ساده ای را برای انجام اشکال زدایی و کنترل کد در اختیار خواهید داشت.

در درس های بعدی به آموزش تکنیک های اشکال زدایی و نحوه کار با کنترل کننده کدها در ابزارهای برنامه نویسی IDE مثل ویژوال استودیو خواهیم پرداخت.

درس ۲۴ : آموزش استفاده از Break Points در عمل Debugging کدهای C#

آموزش استفاده از Break Points در عمل Debugging کدهای C# :

اولین چیزی که بایستی در عملیات اشکال زدایی کدهای C# یا Debugging بایستی بیاموزید، استفاده از BreakPoints ها است. BreakPoints دقیقا همان کاری را انجام می دهد که از نام آن می توان فهمید. BreakPoint نقطه ای در کد شما را تعیین می کند که در آنجا کامپایلر توقف نموده و اجرای برنامه موقتا استپ می شود. در این مکان می توانید به بررسی کدهای خود پرداخته و مقادیر متغیرها و عبارات را در برنامه چک کنید.

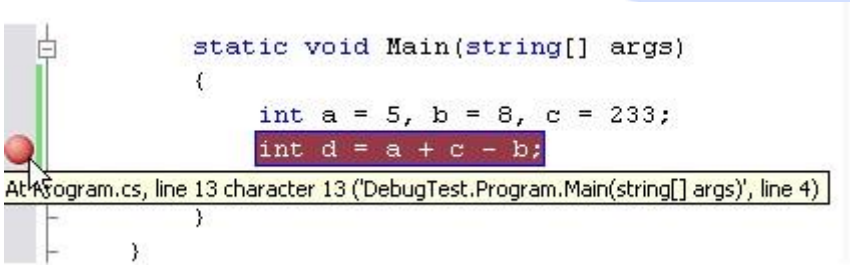
برای قرار دادن BreakPoint در محیط ویژوال استودیو، بایستی بر روی لبه کناری برنامه و هم ردیف با کدی که می خواهید اشکال زدایی کنید، کلیک راست نموده و برنامه برای شما یک دایره قرمز رنگ به نشانه BreakPoint قرار می دهد.

برای درک بهتر، قطعه کد زیر را در محیط ویژوال استودیو کپی کرده و در کنار آن یک BreakPoint ایجاد نمایید :

```
namespace DebugTest
{
  class Program
  {
    static void Main(string[] args)
    {
      int a = 5, b = 8, c = 233;
      int d = a + c - b;
      Console.WriteLine(d);
    }
  }
}
```

کد مثال فوق، کد بسیار ساده ای بوده و شما حتی با یک ماشین حساب می توانید مقدار متغیر را در هر لحظه حساب کرده و کد خود را چک نمایید. اما در پروژه های بزرگ، انجام این کار به صورت دستی تقریبا غیر ممکن است.

همانند تصویر زیر، یک BreakPoint در کنار خط کد نشان داده شده، ایجاد نمایید :



اکنون شما آماده اید تا اولین عملیات اشکال زدایی صفحات یا Debugging خود در C# را انجام دهید. با زدن دکمه F5 برنامه را اجرا نمایید. چیزی که رخ خواهد داد این است که برنامه مثل حالت طبیعی اجرا شده و زمانی که به خط کد دارای Breakpoint برسد، عملیات پردازش برای چک کردن کد متوقف می شود. در کد فوق،

متغیرهای a و b و c دارای مقدار هستند، ولی در خط مشخص شده متغیر d هنوز مقداردهی نشده و مقدار پیش فرض برای integer یعنی صفر را دارا می باشد. پس از اجرای این خط کد است که متغیر d مقدار جدید خود را کسب می کند.

حالا می رسیم به قسمت هیجان انگیز برنامه، موس را بر روی نام متغیرهای مختلف کد ببرید. یک tooHip باز می شود که به شما اعلام می کند هر متغیر در آن لحظه، دارای چه مقداری است.

همان طور که اشاره کردیم، متغیر d تا اجرای این خط، دارای مقدار پیش فرض صفر است. اما با حرکت و به جلو رفتن اجرای برنامه، می توان مقدار آن را تغییر داد. در درس بعدی، به آموزش نحوه حرکت در کدهای برنامه و بررسی بیشتر عملیات Debugging خواهیم پرداخت.

درس ۲۵ : آموزش حرکت بین کدها در هنگام Debug برنامه های C#

آموزش حرکت بین کدها در هنگام Debug برنامه های C# :

در این درس، به آموزش نحوه حرکت بین کدها در زمان Debug برنامه های C# خواهیم پرداخت. برای این منظور، برنامه ای با کد زیر را نوشته ایم :

```
namespace DebugTest
{
  class Program
  {
    static void Main(string[] args)
    {
      int a = 5;
      a = a * 2;
      a = a - 3;
      a = a * 6;
      Console.WriteLine(a);
```

```
    }
  }
}
```

کد مثال فوق، چندین بار بر روی متغیر a عملیات ریاضی انجام داده و سپس نتیجه نهایی را در خروجی چاپ می کند. همان گونه که در درس قبل نشان دادیم، در کنار خطی اولی که متغیر a تعریف و مقداردهی شده است، یک Breakpoint قرار دهید. سپس پروژه را اجرا نمایید. در خط اول ای که Breakpoint قرار داده اید، اجرای برنامه متوقف شده و می توانید با بردن موس بر روی متغیر a، مقدار آن را در لحظه مشاهده نمایید. در این مرحله، متغیر a دارای مقدار پیش فرض متغیرهای عددی یعنی صفر است، زیرا هنوز کد آن خط اجرا نشده است. اما بیایید کمی روند را تغییر دهیم.

از منوی Debug ، گزینه "Step over" را انتخاب کرده و یا دکمه F10 را فشار دهید. در این حالت، خط کد بعدی برنامه اجرا شده و می توانید با حرکت موس بر روی کد ببینید که اکنون متغیر a دارای مقدار است. مجددا با زدن دکمه F10 به حرکت در خط های کد ادامه داده و هر خط را یک به یک اجرا کرده و مقدار متغیر آن را در آن خط بسنجید، تا این که به انتهای کد برسید.

مثال فوق بسیار ساده ولی در عین حال کاربردی بود، اکنون می توانیم وارد مراحل پیشرفته تر بشویم.

در کد مثال زیر، یک برنامه پیچیده تر از مثال قبلی طراحی کرده ایم. در کد مثال قبل، جریان حرکت برنامه بسیار ساده و خطی بود. اما اگر کد ما در بین توابع و کلاس های مختلف حرکت کند، چه؟ برای این منظور به کد مثال زیر دقت کنید :

```
namespace DebugTest
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
        int a = 5;
        int b = 2;
        int result = MakeComplicatedCalculation(a, b);
        Console.WriteLine(result);
    }

    static int MakeComplicatedCalculation(int a, int b)
    {
        return a * b;
    }
  }
}
```

در خط اول متد Main برنامه یک Breakpoint قرار داده و آن را اجرا نمایید. سپس از دکمه F10 برای حرکت بین خطوط برنامه و اجرای کدها استفاده کنید.

همانطور که مشاهده خواهید کرد، برنامه از روی کد فراخوانی تابع بدون توجه عبور می کند. این حالت روش پیش بینی فرض Debugger است. اما بار دیگر اجرای کد را از اول شروع کرده و هنگام رسیدن به خط کد فراخوانی تابع Make Calculation()، از منوی Debug گزینه Step into را انتخاب کرده یا کلید میانبر F11 را فشار دهید. در این هنگام، برنامه وارد عملیات فراخوانی تابع شده و بدنه تابع را به صورت کامل پیمایش می کند. بنابراین با زدن دکمه F11 می توانید ریزعملیات اجرایی برنامه را هم بررسی کنید.

اما اگر وارد یک تابع شدید و در میانه راه خط هستید از مشاهده ادامه روند اجرای تابع صرف نظر کرده و به کد اصلی برنامه باز گردید، می توانید از منوی Debug گزینه "step out" را انتخاب کرده و یا دکمه shift +F11 را همزمان فشار دهید. در این حالت برنامه به ادامه کد پس از تابع بازگشته و به این صورت می توانید هرچند بار که خواستید وارد فراخوانی یک تابع شده و سپس در هنگام لزوم از آن خارج شوید.

درس ۲۶ : آموزش کار با پنجره Tool Window در ویژوال استودیو

آموزش کار با پنجره Tool Window در ویژوال استودیو :

در هنگام انجام عمل اشکال زدایی کدها یا Debugging در ویژوال استودیو، پنجره tool window در قسمت پایین برنامه عوض شده و پنجره های جدیدی ظاهر می شوند (علی رغم این که شما آن ها را خاموش یا غیر فعال کرده باشید). پنجره های جدید باز شده با نام "Locals"، "Watch"، "Call Strack" و "Immediate window" بوده، که مرتبط با عملیات Debugging هستند. در این بخش به بررسی هریک از ابزارها خواهیم پرداخت.

پنجره Locals :

این پنجره، ساده ترین ابزار این بخش می باشد. هنگامی که برنامه به یک Breakpoint می رسد، تمامی متغیرهای محلی کد یا local variables در این پنجره لیست شده و به شما امکان می دهد تا با یک نگاه سریع اطلاعاتی مثل نام، نوع و مقدار آنها را مشاهده نمایید. شما حتی می توانید بر روی نام متغیر مورد نظر خود در جدول کلیک نموده و با انتخاب آیتم "Edit Value"، به متغیر انتخاب شده یک مقدار جدید بدهید. این کار امکان امتحان کردن کدهای برنامه را در شرایط مختلف دیگر می دهد.

پنجره Watch :

پنجره Watch Window تا حدودی شبیه پنجره Locals است. با این تفاوت که در این پنجره می توانید تصمیم بگیرید کدام یک از متغیرها به عنوان محلی local یا سراسری global تعیین شوند. شما می توانید متغیر مورد نظر خود را به محل درگ کردن از کد صفحه، یا پنجره Locals و یا نوشتن نام آن در آخرین خط خالی، تحت نظر گرفته و به پنجره Watch اضافه کنید. متغیرهای لیست Watch تا زمانی که آن ها را حذف نکنید، در پنجره باقی خواهند ماند، اما مقدارشان فقط در زمانی که وارد محدوده کد مرتبط با آن ها بشوید، به روز می شود. برای مثال، یک متغیر متعلق به تابع A در زمانی که درون کد تابع B باشید، Update نمی شود. همانند پنجره Locals می توانید بر روی متغیر مورد نظر خود کلیک راست کرده و با انتخاب گزینه "Edit Value"، مقدار آن را تغییر دهید.

پنجره Call Stack :

پنجره Call Stack ، سلسله مراتب یا hierarchy اجرای توابع برنامه را نشان می دهد. برای مثال اگر تابع A تابع B را فراخوانی کرده و تابع B تابع C را فراخوانی می کند، این ارتباطات را می توانید در پنجره Call stack مشاهده کرده و بر روی کد هر یک از تابع ها در صورت تمایل پرش کنید. شما همچنین می توانید ببینید چه پارامترهایی به هر تابع پاس داده می شوند.

کد مثال های این درس، بسیار ساده بودند و به راحتی می توانید مسیر پیمایش تابع را مشاهده کنید. اما در پروژه های بزرگ، فهمیدن ترتیب اجرای توابع و پارامترهای هریک از آن ها، امر ضروری بوده که با Call stack می توان انجام داد.

پنجره Immediate Window :

پنجره Immediate Window می تواند کاربردی ترین پنجره عملیات Debug باشد. این پنجره به شما امکان می دهد، بخش های مورد نظر خود را از کد برنامه انتخاب کرده و اجرا نمایید. همچنین می توانید مقادیر متغیرهای آن ها را چک کرده و یا تغییر دهید.

فقط کافی است کد مورد نظر خود را در پنجره نوشته و با زدن دکمه Enter آن را اجرا نمایید. نام هر متغیر را تایپ نموده و مقدار آن را در خروجی چاپ کنید. مقدار متغیر مورد نظر خود را با نوشتن a=5 تغییر داده و همان لحظه نتیجه تغییر را در کد مشاهده کنید. پنجره Immediate Window همانند یک ترمینال در C# است، به محض نوشتن کد یا تغییر در برنامه، خروجی را مشاهده خواهید کرد.

درس ۲۷ : آموزش پیشرفته تر کار با Breakpoint در Debug کدهای C#

آموزش پیشرفته تر کار با Breakpoint در Debug کدهای C# :

در درس های قبل، یک Breakpoint ساده را ایجاد کردیم. اما امکانات بسیار بیشتری در هنگام کار با Breakpoint ها، به خصوص در محیط ویژوال استودیو وجود دارد. البته به نظر می رسد، مایکروسافت برخی از

این قابلیت ها را در نسخه های Express ویژوال استودیو، غیر فعال کرده است، اما این امکانات در نسخه فوق همچنان در دسترس است.

قابلیت Condition :

این Condition به شما امکان می دهد تا یک شرط را تعیین کرده و زمانی که آن شرط درست یا true شد یا مقدارش تغییر کرد، breakpoint رخ دهد. این قابلیت، در زمانی که با کدهای سطح بالاتر در تعامل هستید، بسیار کاربردی است. برای مثال هنگامی که می خواهید عملیات اجرای برنامه تحت شرایط خاصی متوقف شود. برای مثال، فرض کنید که یک حلقه loop دارید که تا قبل از رسیدن به کد مورد نظر، چندین بار تکرار می شود. در چنین شرایطی می توانید با تعیین یک شرط یا Condition مورد نظر و اضافه کردن آن به breakpoint، شرایط را کنترل کنید. مثال زیر را برای نشان دادن قابلیت Condition ایجاد کرده ایم :

```
static void Main(string[] args)
{
  for(int i = 0; i < 10; i++)
    Console.WriteLine("i is " + i);
}
```

Breakpoint را برروی خط کدی که خروجی را بر روی Console نمایش می دهد، قرار دهید. سپس برنامه را اجرا کنید. Breakpoint هر بار که حلقه تکرار می شود، رخ می دهد. اما این چیزی نیست که ما دنبال آن هستیم. ممکن است بخواهیم breakpoint فقط در زمانی که i برابر با ۴ است (دفعه ۵ ام تکرار حلقه) فعال شود. بنابراین شرط i==4 را به Breakpoint اضافه می کنیم.

اکنون Breakpoint با یک دایره سفید درون آن نشان داده شده و می گوید که برای اجرا دارای یک شرط است و زمانی اجرا می شود که مقدار متغیر i برابر ۴ شود.

شما همچنین می توانید گزینه "has changed" را اضافه کرده تا به debugger بگویید فقط در زمانی که نتیجه عبارت فوق تغییر کرد، breakpoint را فعال کند، مثلا وقتی که مقدار آن از true به false عوض شد.

قابلیت Hit Count :

با استفاده از قابلیت Hit Count می توانید شرط دیگری را برای اجرای Breakpoint تعیین کرده که بر حسب تعداد دفعاتی خواهد بود که Breakpoint در کد روی می دهد.

برای مثال، می توانید تصمیم بگیرید که Breakpoint، تعداد دفعات مشخصی در کد تکرار نشده باعث توقف اجرای برنامه شود. تنظیمات مختلفی برای این شرط وجود دارد که می توانید در هنگام کار عملی، آن ها را در سطح برنامه تغییر دهید.

قابلیت When hit :

به وسیله این پنجره می توان رفتار متفاوتی را برای برنامه در هنگام رسیدن Breakpoint تعیین کنید. این امکان، بیشتر در مواردی کاربرد دارد که نمی خواهید برنامه با رسیدن به Breakpoint متوقف شده، به جای آن یک پیام خاصی منتشر شده یا یک macro اجرا شود. این قابلیت به شما امکان می دهد تا یک پیام دلخواه را تعیین کرده و اطلاعات خاصی را در برنامه روند اجرا برنامه چاپ کنید. در کدهای پیشرفته تر، می توانید یک macro را تعیین نموده تا به محض رسیدن برنامه به نقطه Breakpoint، اجرا شود.

درس ۲۸ : آموزش کار با Enumeration در زبان C#

آموزش کار با Enumeration در زبان C# :

Enumeration ها، مجموعه ای از مقادیر با نام (named value) هستند که متناظر با مجموعه ای از اعداد، معمولا از نوع integer می باشند. Enumeration ها در موارد مختلفی کاربرد دارند، مثلا زمانی که می خواهید قادر باشید از بین مجموعه ای از مقادیر ثابت، به راحتی مقدار مورد نظر خود را انتخاب کرده و آن را متناظر با یک مقدار عددی قرار دهید.

همان طور که در کد مثال های عملی این درس خواهید دید Enumeration ها در بالای کلاس ها و در درون namespace تعریف می شوند. بنابراین می توان از هر Enumeration در کل namespace جای برنامه، استفاده کرد.

در کد زیر، یک مثال ساده از نوع تعریف Enumeration ها در زبان C# نشان داده شده است :

```
public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }
```

تمامی مقادیر موجود در Enumeration فوق، متناظر با یک مقدار عددی هستند. اگر مقادیر متناظر Enumeration ها را به صورت دستی تعیین نکنید، به طور پیش فرض اولین مقدار متناظر با ۰، مقدار بعدی متناظر با ۱ و به همین ترتیب خواهند بود.

کد زیر، نحوه ارتباط پیش فرض هر یک از آیتم های Enumeration را نشان داده و به شما نحوه استفاده از یک مقدار خاص بر حسب یک Enumeration را یاد می دهد :

```
using System;

namespace ConsoleApplication1
{
  public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }

  class Program
  {
    static void Main(string[] args)
    {
      Days day = Days.Monday;
      Console.WriteLine((int)day);
      Console.ReadLine();
    }
  }
}
```

خروجی کد مثال فوق صفر خواهد بود، زیرا مقدار Monday مستقیما با مقدار ۰ متناظر خواهد بود. البته توسط کد زیر می توانید عدد متناظر با Enumeration ها را به مقدار دلخواه تغییر دهید :

```
public enum Days { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }
```

اگر کد فوق را اجرا کنید، خواهید دید که این بار Monday به جای صفر متناظر با یک خواهد بود. سایر مقادیر Enumeration را هر یک به ترتیب دارای یک مقدار بیشتر از ۱ خواهند بود.

نکته : شما می توانید هر عضو Enumeration را به یک عدد خاص و لزوما غیر پشت سر هم نسبت دهید.

به دلیل ارتباط مستقیم هر عضو Enumeration با عدد متناظر آن، می توانید به وسیله اعداد مقدار هر عضو Enumeration مورد نظر خود را بخوانید. به صورت کد زیر :

```
Days day = (Days)5;
Console.WriteLine(day);
Console.ReadLine();
```

قابلیت دیگر Enumeration ها این است که به وسیله حلقه های تکرار، می توانید مقادیر رشته ای (string) یک Enumeration را در خروجی نشان دهید. کد مثال قبل را به کد زیر تغییر دهید :

```
static void Main(string[] args)
{
  string[] values = Enum.GetNames(typeof(Days));
  foreach(string s in values)
    Console.WriteLine(s);

  Console.ReadLine();
}
```

کلاس Enum Class دارای گروهی از متدهای کاربردی است که می توانید آن ها را برای کار با Enumeration ها استفاده کنید.

درس ۲۹ : آموزش مدیریت خطا Exception Handling در C#

آموزش مدیریت خطا Exception Handling در زبان C# :

در هر برنامه ای، همواره خطاهایی رخ داده و اجرای امور با مشکل مواجه می شود. در زبان C#، کامپایلر هوشمند و کارآمدی در اختیار ما قرار داده شده که به کمک آن می توانیم از برخی اشتباه رایج جلوگیری کنیم. البته که برنامه همه خطاهای کد را نخواهید دید و در چنین مواردی، چهارچوب کاری .NET.، یک خطا یا Exception اعلام کرده تا به ما بگوید جایی در کد دارای اشکال است. در درس های قبل و در بخش آموزش کار با آرایه ها Arrays در C#، نشان دادیم چگونه با وارد کردن آیتم هایی بیش از تعداد تعیین شده برای آرایه، می توان برنامه را با خطا مواجه کرد.

بیایید نگاهی به کد مثال بخش آموزش آرایه ها در C# بیندازیم :

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] numbers = new int[2];
```

```
        numbers[0] = 23;
        numbers[1] = 32;
        numbers[2] = 42;

        foreach(int i in numbers)
          Console.WriteLine(i);
        Console.ReadLine();
    }
  }
}
```

کد فوق را اجرا نموده و مشاهده کنید که برنامه خطا اعلام می کند. خطای کد این است که ما یک آرایه با دو عضو را تعریف نموده، ولی سعی داریم تا ۳ عضو را در آن قرار دهیم.

وقتی در محیط یک IDE مثل Visual Studio کد فوق را اجرا کنید، برنامه چند راه حل یا توضیح را درباره خطای رخ داده، اعلام می کند. اما اگر بخواهید برنامه را با دابل کلیک بر روی فایل EXE آن اجرا کنید، یک خطای نامفهوم برایتان رخ داده و اجرای فایل متوقف می شود.

اگر احتمال می دهید در بخشی از کد خطا رخ خواهد داد، بایستی بتوانید ان را مدیریت کنید. اینجاست که Exception در C# به کمک ما می آید. کد مثال فوق را کمی تغییر داده و به صورت زیر بازنویسی کرده ایم :

```
int[] numbers = new int[2];
try
{
  numbers[0] = 23;
  numbers[1] = 32;
  numbers[2] = 42;
```

```
  foreach(int i in numbers)
    Console.WriteLine(i);
}
catch
{
  Console.WriteLine("Something went wrong!");
}
Console.ReadLine();
```

بیایید با یکی از کارآمدترین ابزارهای کدیریت خطا در C#، یعنی ساختار try-catch آشنا شویم. بار دیگر برنامه را اجرا کرده و تفاوت را با سری قبل مشاهده کنید. اما به جای این که ویژوال استودیو یا ویندوز نوع خطا را به ما اعلام کند، خودمان می توانیم پیام آن را تعیین کرده یا به دلخواه تنظیم کنیم. به صورت کد زیر :

```
catch(Exception ex)
{
  Console.WriteLine("An error occured: " + ex.Message);
}
```

همانطور که مشاهده می کنید، یک بخش جدید را به ساختار دستوری try-catch اضافه کرده ایم. به وسیله کد اضافه شده، می خواهیم بدانیم کدام خطا در برنامه رخ داده است و در این مثال از کلاس اصلی خطاها، یعنی Exception استفاده کرده ایم. با انجام کد فوق، ما اطلاعاتی راجع به خطاهای رخ داده کسب خواهیم کرد و به وسیله خاصیت Message-Property یک توضیح قابل فهم را درباره خطا مشاهده می کنیم.

همانطور که گفتیم Exception رایج ترین نوع خطا در هنگام اجرای برنامه های C# است. قوانین خطایابی در زبان متنی شارپ به ما می گوید همواره بایستی از EXCEPTION که کمترین احتمال رخ دادن را دارند، در کد

خود استفاده کنیم. اما در مثال این درس، ما در اصل می دانستیم چه خطایی ممکن است در برنامه رخ دهد. اما چگونه؟

به دلیل این که ویژوال استودیو خطای مدیریت نشده را به ما اعلام کرد. اما اگر شک دارید، خطا کدام است، معمولا شرح خطا یا Exception توسط برنامه بیان می شود. راه دیگر برای یافتن نوع خطا، استفاده از کلاس Exception Class است. برای این منظور کد مثال قبل را به صورت زیر تغییر دهید :

```
Console.WriteLine("An error occured: " + ex.GetType().ToString());
```

نتیجه کد فوق، همانطور که می توان پیش بینی کرد خطای "مقدار خارج از محدوده" یا Index Out Range Exception می باشد. ما بایستی این خطا را مدیریت یا handle کنیم. اما در هر زمان ، می توان بیش از یک خطا را در کد مدیریت نمود و در صورت رخ دادن هر خطا تصمیم لازم را تعیین کرد. بنابراین کد مثال قبل را به صورت زیر تغییر دهید :

```
catch(IndexOutOfRangeException ex)
{
  Console.WriteLine("An index was out of range!");
}
catch(Exception ex)
{
  Console.WriteLine("Some sort of error occured: " + ex.Message);
}
```

همانگونه که در کد فوق تعیین کرده ایم، برنامه ابتدا به دنبال خطاهای Index Out Range Exception می گردد. اگر از روش دوم برای خطایابی استفاده کنیم، ساختار Catch توسط کلاس Exception Class مشکل کد را پیدا خواهد کرد زیرا تمامی Exception ها از این کلاس مشتق می شوند. به عبارت دیگر، در خطایابی کدها بایستی همواره از خطایی که احتمال رخ دادن آن بیشتر است، اول استفاده کنیم.

چیز دیگری که بایستی در مورد مدیریت خطاها Exception Handling بایستی بدانید، بلوک کد نهایی یا finally block است. بلوک finally را می توان به انتهای ساختار دستوری try-catch اضافه کرده یا بر حسب نیاز، به صورت جداگانه به کار برد.

کد تعیین شده در بخش finally در هر صورت اجرا خواهد شد، چه Exception رخ دهد یا خیر. بنابراین محل خوبی برای قرار دادن کدهایی مثل قطع ارتباط با فایل های برنامه یا تخریب اشیایی که دیگر به آن ها نیاز نداریم، می باشد. از آنجایی که مثال های قبلی ارائه شده، نسبتا ساده بودند، بنابراین به عملیات پاکسازی یا Clean up خاصی نیاز نداشتیم و خود کننده خودکار C# یعنی Garbage Collector کارهای لازم را انجام می داد.

اما گاهی اوقات در کدهای گسترده تر، نیاز به قطعه کد نهایی یا finally داریم که مثال زیر، نحوه استفاده از این امکان را به صورت عملی نشان داده است :

```
int[] numbers = new int[2];
try
{
   numbers[0] = 23;
   numbers[1] = 32;
   numbers[2] = 42;

   foreach(int i in numbers)
      Console.WriteLine(i);
}
catch(IndexOutOfRangeException ex)
{
   Console.WriteLine("An index was out of range!");
}
```

```
catch(Exception ex)
{
  Console.WriteLine("Some sort of error occured: " + ex.Message);
}
finally
{
  Console.WriteLine("It's the end of our try block. Time to clean up!");
}
Console.ReadLine();
```

اگر کد مثال فئق را اجرا نمایید، مشاهده خواهید کرد که هم دستورات بخش خطا یا Exception و هم دستورات بخش نهایی یا finally code اجرا می شوند. اگر خط کدی که عدد ۴۲ را به آرایه مثال اضافه می کند، پاک نمایید، خواهید دید که فقط دستورات بخش finally انجام می شود، زیرا دیگر خطایی در کد وجود ندارد.

مشئله مهم دیگری که بایستی در مورد exception ها بدانید، نحوه تاثیرگذاری خطاها بر روی متدها و توابعی است که exception در آن ها رخ می دهد. همه خطاهای غیر مدیریت شده در برنامه شما، لزوما از بین برنده برنامه نبوده و باعث خروج استمراری برنامه نمی شوند. اما وقتی هم که باعث توقف کامل برنامه نمی شوند، نبایستی انتظار داشته باشید که بقیه کد متد یا تابع دارای exception اجرا شود.

به عبارت دیگر، اگر شما exception را مدیریت کنید، خطاهای کد بعد از بلوک دستوری try اجرا می شوند، نه دستورات درون خود آن. در مثال فوق، حلقه ای که مقادیر آرایه array را در خروجی چاپ می کند، هرگز اجرا نخواهد شد، زیرا به دلیل بروز exception یا خطا در کد try، برنامه به خط اول بعد از بلوک try و بخش های Catch یا finally پرش می کند. اما خط آخر که در آن برای جلوگیری از خروج ویندوز از برنامه، منتظر دریافت ورودی از کاربر هستیم (Consule.ReadLine)، همواره اجرا شود. بنابراین در زمان طراحی ساختار های دستوری try-catch بایستی این نکته را مد نظر داشته باشید.

درس ۳۰ : آموزش کار با Structs در زبان C#

آموزش کار با Structs (ساختارها) در زبان C# :

ساختارها یا Structs یک جایگزین سبک حجم (light weight) برای کلاس ها Class در زبان C# هستند. دلیل بیان این مسئله، کمی نیازمند بحث تکنیکال است، اما اگر بخواهیم خلاصه بیان کنیم، نمونه های ساخته شده از یک کلاس instances Class (بر روی حافظه) در یک ساختار درختی یا heap قرار می گیرند، در حالی که نمونه های ساخته شده از یک Structs به صورت Stack ذخیره می شوند.

علاوه بر این، در ساختار Struct بر عکس Class ها، با مرجع یا refrence به یک struct کار نمی کنید، بلکه مستقیما به نمونه ساخته شده از structs دسترسی دارید. این همچنین به این معنی است که وقتی شما یک struct را به عنوان پارامتر به تابع پاس می دهید، به صورت call by value فرستاده می شود نه call by refrence. در مورد تفاوت این روش در درس ارسال پارامترها به توابع زبان C# کامل صحبت کردیم. جهت دریافت اطلاعات بیشتر به بخش مذکور رجوع نمایید.

نکته : زمانی که می خواهید دیتا ساده تری را نشان دهید، بهتر از ساختار Struct استفاده، مخصوصا در مواقعی که می خواهید نسخه های زیادی از آن ها را مقداردهی کنید.

مثال های زیادی در چهارچوب کاری .NET وجود دارد که مایکروسافت از ساختار struct به جای کلاس ها استفاده کرده، مثل اشیای Point، Rectangle و یا Color struct.

در مرحله اول، یک مثال عملی از نحوه تعریف و استفاده از struct ها را به شما نشان داده و سپس درباره محدودیت های استفاده از ان ها به جای کلاس ها صحبت خواهیم کرد :

```
static void Main(string[] args)
{
    Car car;

    car = new Car("Blue");
    Console.WriteLine(car.Describe());
```

```
    car = new Car("Red");
    Console.WriteLine(car.Describe());

    Console.ReadKey();
  }
}

struct Car
{
  private string color;

  public Car(string color)
  {
    this.color = color;
  }

  public string Describe()
  {
    return "This car is " + Color;
  }

  public string Color
  {
    get { return color; }
```

```
        set { color = value; }
    }
}
```

اگر به مثال خوب دقت کرده باشید، متوجه می شوید این همان کد مثالی است که در بخش معرفی کلاس ها در C# به کار بردیم، با این تفاوت که از struct به جای class در بخش دوم ان استفاده کرده ایم. این مثال به خوبی نشان می دهد که این دو مفهوم چقدر شبیه هم هستند. اما با توجه به توضیحی که در ابتدای درس عنوان کردیم، این دو مفهوم چگونه با هم فرق دارند؟

اول از همه، فیلدها (fields) را در ساختارهای struct نمی توان مقداردهی اولیه کرد، یعنی به وسیله کد متنی کد زیر نمی توان یک member مقدار داد :

```
private string color = "Blue";
```

اگر یک تابع سازنده یا Constructor را برای struct تعیین کنید، بایستی کلیه فیلدها قبل از خروج از Constructor مقداردهی شوند. خود ساختار struct دارای یک تابع سازنده یا default Constructor می باشد، اما اگر به صورت دستی بخواهید تابع سازنده آن را تعیین کنید، بایستی تمامی فیلدها را در تعریف تابع سازنده، مقداردهی کنید. از طرف دیگر این بدین معناست که شما نمی توانید یک تابع سازنده بدون پارامتر را برای struct تعیین نمایید، بلکه تمامی توابع سازنده struct ها حداقل بایستی دارای یک پارامتر باشند. در کد مثال فوق، ما یک مقدار مشخصی را برای فیلد color تعیین نمودیم، که اگر این کار را انجام نمی دادیم، کامپایلر خطا صادر می کرد.

یک struct نمی تواند از سایر کلاس ها یا struct های دیگر به ارث برود (inherit) و کلاس ها هم نمی توانند از struct ها به ارث برده شوند. Structs همچنین از Interface ها پشتیبانی می کنند که بدین معناست که هر struct می تواند Interface های مورد نظر خود را پیاده سازی کند.

درس ۳۱ : آموزش کار با فایل XML در زبان C#

آموزش کار با فایل XML در زبان C# :

XML مخفف عبارت Exteraible Markup Language یا زبان نشانه گذاری قابل توسعه است. زبان XML ، یک فرمت گسترده و رایج برای نقل و انتقال اطلاعات (data) می باشد، بیشتر به این دلیل که به راحتی برای هم انسان و هم ماشین قابل خواندن و درک است.

اگر تاکنون سایتی را به زبان HTML نوشته باشید، XML بسیار برای شما آشنا به نظر خواهد آمد، زیرا درواقع XML یک نسخه سخت گیرانه تر و توسعه داده شده تر از HTML است. XML از تگ ها (tags)، خصوصیات (attributer) و مقادیر (values) تشکیل شده و ساختار کلی کدهای آن، همانند کد مثال زیر است :

```
<users>
 <user name="John Doe" age="42">
 <user name="Jane Doe" age="39">
</user></user></users>
```

همانطور که در کد مثال فوق می توانید مشاهده کنید، XML یک فرمت مناسب برای تعریف اطلاعات یا data بوده و اکثر زبان های برنامه نویسی دارای کلاس ها و توابع خاصی برای کار با XML هستند. زبان C# هم یکی از این زبان های برنامه نویسی است که یک فضای کلاس مخصوص یا namespace، برای کار با XML دارد. نام این namespace در زبان C#، مجموعه System .XML است که تقریبا توانایی کار با قابلیت های مختلف زبان XML را داراست. در درس های بعدی به بررسی استفاده از این کلاس های آماده در زبان C#، هم برای خواندن و هم برای نوشتن اطلاعات به زبان XML خواهیم پرداخت.

درس ۳۲ : آموزش خواندن فایل های XML به وسیله کلاس XMLReader در C#

آموزش خواندن فایل های XML به وسیله کلاس XMLReader در زبان C# :

به طور کلی ۲ متد برای خواندن فایل های XML در زبان C# وجود دارد :

کلاس . xmlDocument

کلاس XMLReader .

کلاس XMLDocument کل محتویات فایل XML را خوانده و در حافظه سیستم قرار می دهد. سپس به شما امکان می دهد به راحتی درون فایل XML به جلو و عقب حرکت کرده و حتی با استفاده از تکنولوژی XPath ، جستجو یا query مورد نظر خود را بر روی فایل انجام دهید.

کلاس XMLReader، گزینه ای سریع تر و کمتر حافظه بر (memory consuming) بر خواندن فایل های XML می باشد. کلاس XMLReader به شما امکان می دهد تا در هر لحظه به وسیله فقط یک المنت، درون محتویات فایل XML حرکت کرده و همزمان مقدار value ها را خوانده و سپس به المنت بعدی در فایل بروید. با انجام اعمال فوق، بدیهی است که برنامه حافظه بسیار کمتری را اشغال می کند، زیرا در هر لحظه فقط مقدار المنت جاری را در خود نگهداری خواهد کرد. علاوه بر این، به دلیل این که می توانید به صورت دستی مقدار هر value را چک کنید، مستقیما به مقادیر مورد نظر خود دسترسی خواهید داشت و این مسئله کار را بسیار سریع تر می کند.

در این درس، بر روی کار با کلاس XMLReader تمرکز کرده و در درس بعدی به آموزش کلاس XMLDocument Class خواهیم پرداخت.

برای آموزش عملی مباحث، بیایید با یک مثال کوچک کار کنیم که در آن یک فایل XML حاوی نرخ های ارز (currency rates) را خواهیم خواند. برای این منظور از یک فایل XML مربوط به بانک مرکزی اروپا استفاده می کنیم. می توانید این فایل را دانلود کرده و از روی هارد دیسک بخوانید. اما درواقع هر دو کلاس XMLReader و کلاس XMLDocument می توانند به خوبی، محتویات یک فایل XML را از روی یک آدرس Url بر روی یک سرور را ه دور همانند فایل های لوکال بخوانند و فایل XML مثال را می توانید از آدرس + آدرس مشاهده نمایید، در کد مثال زیر، بخشی از این فایل را استفاده خواهیم کرد :

```
using System;
using System.Text;
using System.Xml;
```

```
namespace ParsingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlReader xmlReader = XmlReader.Create("http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml");
      while(xmlReader.Read())
      {
        if((xmlReader.NodeType == XmlNodeType.Element) && (xmlReader.Name == "Cube"))
        {
          if(xmlReader.HasAttributes)
            Console.WriteLine(xmlReader.GetAttribute("currency") + ": " + xmlReader.GetAttribute("rate"));
        }
      }
      Console.ReadKey();
    }
  }
}
```

در ابتدای کار یک نسخه جدید از کلاس XMLReader Class را با استفاده از متد Create() ایجاد کرده ایم. این متد چندین overload (مجموعه پارامتر) را می تواند دریافت کند، اما ساده ترین حالت آن ارسال یک Url با آدرس فایل XML مورد نظر برای خواندن است. درون حلقهwhile loop، متد Read() را بر روی نسخه ایجاد

شده از کلاس XMLReader فراخوانی می کنیم. این حلقه، خواننده فایل یا Reader را به المنت بعدی در فایل منتقل کرده و در صورت وجود داشتن المنت ی دیگر، مقدار true را بر می گرداند تا عملیات جستجو همچنان ادامه داشته باشد. با تمام المنت های فایل و صدور مقدار false، اجرای حلقه و خواندن فایل متوقف خواهد شد. در درون حلقه while loop می توانید از خواص (properties) و متدهای (methods) متنوع کلاس XMLReader برای دسترسی به اطلاعات المنت جاری یا مورد نظر خود استفاده کنید.

در مثال فوق، ابتدا خاصیت NodeType را چک کرده تا مطمئن شده فایل دارای المنت (قسمت tag) بوده و همچنین آیا نام آن برابر با "Cub" است یا خیر. همانطور که در فایل XML مثال می توانید ببینید، هر نرخ ارز (currency rate) دارای یک المنت با نام تگ Cube می باشد و این دقیقا چیزی است که ما به دنبال آنیم. به محض پیدا کردن یک المنت Cube ، یک چک فرمت (format check) انجام داده تا ببینیم آیا المنت ما دارای خواص (attributes) بوده و یا خیر و سپس توسط متد ()Get Attribute مقدار دو خاصیت "currency" و "rate" را می خوانیم. در نهایت مقادیر دو خاصیت را در خروجی چاپ کرده و به سراغ المنت بعدی می رویم. خروجی کد مثال فوق، بایستی لیستی از Currency های موجود و نرخ آن ها (rate) باشد.

همانطور که مشاهده کردید روند کار بسیار ساده بود. البته این بیشتر به دلیل این است که ما اطلاعات را به همان ترتیبی که خواندیم، نیاز داشتیم، بدون این که عملیات خاصی بر روی آن ها انجام دهیم. به عبارت دیگر، وقتی لیستی از داده ها را به صورت لیست ترتیبی نمایش دادیم. در درس بعدی، به آموزش نحوه خواندن فایل های XML با استفاده از کلاس XMLDocument Class خواهیم پرداخت و تفاوت کار را مشاهده می کنید.

نکته : بابیستی اطلاع داشته باشید که راه های زیادی وجود داشته که اجرای کد فوق با اشکال رو به رور شود و برنامه خطا یا exception صادر کند. بنابراین بایستی آمادگی مدیریت خطاهای احتمالی را داشته باشید، برای این منظور به درس آموزش مدیریت خطا یا exception در زبان C# بروید.

درس ۳۳ : آموزش خواندن فایل های XML با کلاس XMLDocument در زبان C#

آموزش خواندن فایل های XML با کلاس XMLDocument در زبان C# :

همانطور که در درس قبلی اشاره کردیم، کلاس XMLDocument حافظه بیشتری از کلاس XMLReader مصرف کرده و به طبع کند تر از آن می باشد. اما به هر حال، به چند دلیل، کار با کلاس XMLDocument کمی ساده تر بوده و در برخی موارد نیاز به نوشتن کد کمتری خواهد داشت.

به محض این که محتویات فایل XML را خواندید، می توانید اطلاعات درون آن را به روش درختی (hierarchical) مورد ارزیابی قرار دهید، دقیقا همانند ساختار فایل های XML که در آن هر المنت دارای چندین عنصر فرزند یا child بوده و هر child نیز می تواند چندین عنصر فرزند داشته باشد و به همین ترتیب.

در درس قبلی، اطلاعات را از یک فایل XML مربوط به بانک مرکزی اروپا خواندیم که درباره نرخ های ارز و تغییرات آن ها داده هایی را ارائه می دهد. در این درس هم به انجام این کار خواهیم پرداخت، اما با استفاده از کلاس XMLReader به جای XMLReader.

اطلاعات فایل XML را می توانید از آدرس +آ درس دریافت نموده و داده ای که در آن نیاز داریم، در عنصر <cube> قرار داد. ساختار درختی فایل XML مثال، چیزی همانند کد زیر است :

```
< gesmes:Envelope >
  [other child nodes]
  < Cube >
    < Cube time="2011-04-12" >
      < Cube currency="USD" rate="1.4470"/ >
      < Cube currency="JPY" rate="121.87"/ >
      ...
```

المنت <gesmes:Envelope>، المنت اصلی یا root element فایل است، که می توان توسط خاصیت DocumentElement به آن دسترسی داشت.

در مرحله بعدی، می توانیم با استفاده از خاصیت ChildNodes Collection به عنصرهای فرزند المنت اصلی دسترسی پیدا کنیم. در مثال این درس، قصد داریم تا به عنصرهای فرزند یا child در سه رده یا level زیر المنت اصلی دسترسی پیدا کنیم. این کار را با استفاده از کد مثال عملی زیر می توان انجام داد، دقیقا مشابه کاری که در درس قبل با کلاس XMLReader انجام دادیم :

```
using System;
using System.Text;
using System.Xml;

namespace ParsingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlDocument xmlDoc = new XmlDocument();
      xmlDoc.Load("www.ecb.int/stats/eurofxref/eurofxref-daily.xml");
      foreach(XmlNode xmlNode in
xmlDoc.DocumentElement.ChildNodes[2].ChildNodes[0].ChildNodes)
        Console.WriteLine(xmlNode.Attributes["currency"].Value + ": " +
xmlNode.Attributes["rate"].Value);
      Console.ReadKey();
    }
```

```
    }
}
```

همانطور که در کد مثال مشاهده می کنید، ما به عنصر Cube nodes با پایین تر رفتن از ساختار درختی ChildNodes hierarchy دسترسی پیدا کرده ایم. از المنت اصلی فایل که DocumentElement نام دارد، سراغ المنت سوم فرزند child node (دارای اندیس بر پایه صفر) را گرفته ایم. سپس به دنبال اولین فرزند این المنت یا first child node گشته و در نهایت کلیه عناصر فرزند این عنصر یا Collection of child nodes را درخواست کرده ایم.

بدیهی است کد فوق به دلیل این که ما با ساختار فایل XML مثال آشنا هستیم، کاربرد داشته و برای مراحل بعدی در آینده چندان انعطاف پذیر و مناسب نیست.

به هر حال روشی که شما برای پیدایش یک فایل XML به کار خواهید برد، بستگی زیادی به سورس فایل XML و نوع اطلاعاتی که دنبال آن هستید، دارد.

کد مثال این درس، تنها در صورتی که حجم محدودی از اطلاعات داشته باشیم، کار خواهد کرد اما برای اهداف بزرگ تر، نیازمند نوشتن کد بهتری هستیم تا خوانایی برنامه افزایش پیدا کند.

به دلیل این که ما دارای یک گره یا node به نام Currency rate هستیم و می توانیم به دو خاصیت یا properties آن دسترسی داشته و آن را در خروجی چاپ کنیم، کد فوق مثال خوبی است همانند درس قبل.

درس ۳۴ : آموزش خواندن فایل های XML با کلاس XmlNodes در C#

آموزش خواندن فایل های XML با کلاس XmlNodes در C# :

در درس قبلی، از کلاس XMLDocuments برای خواندن فایل XML استفاده کردیم. یک کلاس جدید را در مثال درس قبل با نام کلاس XmlNode معرفی کردیم که برای تجزیه و خواندن فایل های XML بسیار ضروری است. به طور کلی، فایل XML به یک XmlNode که المنت اصلی یا root فایل است، تجزیه شده و سپس به وسیله آن می توانید با استفاده از خاصیت childNodes به عناصر فرزند المنت اصلی دسترسی داشته باشید.

همچنین، کلاس XmlNode امکان دسترسی به اطلاعات بسیار دیگری از جمله نام تگ ها یا tag name، خواص یا attributes، متن درون تگ ها یا inner text وجود ساختار XML را نیز به ما می دهد.

در این درس قصد داریم تا توضیحی مختصر درباره برخی از جنبه های جالب کلاس XmlNode ارائه دادیم، زیرا داشتن اطلاعات درباره XmlNode به عنوان یکی از جنبه های کلیدی در هنگام خواندن فایل های XML توسط کلاس XMLDocument، بسیار مهم است.

در مثال های این درس، از المنت DocumentElement بسیار استفاده خواهیم کرد و از آنجایی که این عنصر از نوع XMLElement بوده و خود XMLElement از کلاس XmlNode به ارث رفته است، درواقع این نوع دوستی تقریبا یکسان هستند.

خاصیت Name Property به سادگی نام عنصر یا node را اعلام می کند. برای مثال، خروجی کد زیر مقدار "user" خواهد بود :

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<user name=\"John Doe\">A user node</user>");
Console.WriteLine(xmlDoc.DocumentElement.Name);
Console.ReadKey();
```

خاصیت InnerText Property هم متن بین تگ باز و بسته هر tag را همانند کد زیر استخراج می کند :

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<test>InnerText is here</test>");
Console.WriteLine(xmlDoc.DocumentElement.InnerText);
Console.ReadKey();
```

خاصیت InnerXml اندکی مشابه خاصیت InnerText می باشد، اما در حالی که InnerText property هرگونه XML که درون آن می باشد را حذف می کند، InnerXml property صراحتا این کار را انجام نمی دهد. مثال پایین این تفاوت را نشان می دهد:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<users><user>InnerText/InnerXml is here</user></users>");
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);
Console.WriteLine("InnerText: " + xmlDoc.DocumentElement.InnerText);
Console.ReadKey();
```

خاصیت OuterXml Property نیز همانند InnerText عمل می کند، با این تفاوت که کدهای XML را علاوه بر متن آن ها نیز نشان می دهد. کد زیر می تواند تفاوت دو خاصیت را نشان دهد :

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<users><user>InnerText/InnerXml is here</user></users>");
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);
Console.WriteLine("OuterXml: " + xmlDoc.DocumentElement.OuterXml);
Console.ReadKey();
```

همچنین در کد مثال زیر، با خواص یا attributes تگ ها نیز کار کرده ایم :

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<user name=\"John Doe\" age=\"42\"></user>");
if(xmlDoc.DocumentElement.Attributes["name"] != null)
  Console.WriteLine(xmlDoc.DocumentElement.Attributes["name"].Value);
if(xmlDoc.DocumentElement.Attributes["age"] != null)
  Console.WriteLine(xmlDoc.DocumentElement.Attributes["age"].Value);
```

```
Console.ReadKey();
```

درس ۳۵ : آموزش استفاده از XPath به همراه کلاس XmlDocument در C#

آموزش استفاده از XPath به همراه کلاس XmlDocument در زبان C# :

در درس های قبل، از کلاس XnlDocument برای استخراج اطلاعات از فایل xml استفاده کردیم. ما این کار را با چندین بار فراخوانی خاصیت ChildNodes انجام دادیم که به علت ساده بودن فایل به کار رفته در مثال، کار ساده ای بود. روش فوق چندان خوانایی مناسبی برای کد ما ندارد، بنابراین در این درس به آموزش راه حلی دیگری خواهیم پرداخت.

تکنولوژی که در این درس از آن استفاده خواهیم کرد، XPath نام دارد که توسط همان کنسرسیومی که XML را گسترش داده، ایجاد گردیده است. XPath درواقع یک زبان کامل جستجو یا query language است با قابلیت های گسترده، اما از آنجا که این مجموعه، آموزش XPath نیست، ما فقط به بررسی برخی از query های پرکاربرد خواهیم پرداخت. اما به هر حال، حتی در ساده ترین مثال ها نیز ، XPath یک ابزاری قدرتمند است، همانطور که در مثال های این درس مشاهده خواهید کرد.

کلاس XMLDocument دارای چندین متد (methods) بوده که XPath query را به عنوان پارامتر ورودی دریافت کرده و نتایج حاصله را به صورت گروهی از XMLNode بر می گرداند. در این درس، به بررسی دو متد مختلف خواهیم پرداخت :

- متد Select Single Node() که یک XmlNode تنها را بر حسب Xpath quert تعیین شده، جهت آن بر می گرداند.
- متد SelectNodes() که مجموعه ای از XmlNode ها را بر حسب Xpath query تعیین شده برای آن، بر می گرداند (به صورت یک مجموعه ای از XmlNode objects).

ما هر دو متد معرفی شده را امتحان خواهیم کرد، اما بر خلاف درس های قبل که از یک فایل XML مرتبط با نرخ های ارز استفاده کردیم، از یک سورس جدید XML برای مثال هایمان بهره می گیریم. فردهای RSS feeds

سندهای XML بسیار خوبی هستند که به شیوه خاص خود طراحی شده و به کاربران اجازه می دهند تا اخبار و اطلاعیه های مختلف را با روش مورد نظر خود جستجو کرده و بخوانند.

برای مثال های این درس از یک فید RSS Feed مربوط به سایت CNN از آدرس +آدرس استفاده خواهیم کرد که اخبار مختلفی از نقاط مختلف جهان را ارائه می دهد. اگر صفحه اشاره شده فوق را در مرورگر خود باز کنید، مرورگر شما اخبار را به شیوه ای جذاب برایتان نمایش داده و به سرعت می توانید تمامی Feed های جدید را با عناوین آن ها مرور کنید. اما گول این سیستم را نخورید! در پشت پرده، این صفحه یک فایل XML خاص است که اگر دکمه "View Source" را بر روی صفحه بزنید، کلیه محتویات آن را به صورت XML نشان می دهد. در کد پشت صفحه خواهید دید که محتویات XML یک المنت اصلی به نام "rss" دارند. هر المنت "rss" یک یا چندین المنت فرزند "channel" داشته که درون هر یک از آن ها می توانید اطلاعات لازم درباره هر feed را مشاهده کنید. همچنین عناصر nodes "item" مشروح اخبار که درواقع به دنبال آن ها هستیم را در بر می گیرند.

در مثال این درس، ما از متد SelectSingle.Node() برای دریافت عنوان یا title یک فید استفاده خواهیم کرد. اگر به سورس کد فایل XML نگاهی بیاندازید، خواهید فهمید که المنت <title> یک المنت فرزند برای المنت <channel> بوده که خود المنت عنصر اصلی یا root فایل یعنی <rss> است. Query مورد نیاز را به صورت زیر می توان در زبان Xpath نوشت :

```
//rss/channel/title
```

در کد فوق، به سادگی نام یا name هر المنتی که به دنبال آن هستیم را نوشته و آن ها را با کاراکتر (/) از هم جدا کرده ایم. کاراکتر (/) تعیین می کند که هر المنت بعد از آن، فرزند یا child المنت قبل از آن می باشد. استفاده از زبان XPath به سادگی در کد مثال زیر نشان داده شده است :

```
using System;
using System.Text;
```

```
using System.Xml;


namespace ParsingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlDocument xmlDoc = new XmlDocument();
      xmlDoc.Load("rss.cnn.com/rss/edition_world.rss");
      XmlNode titleNode = xmlDoc.SelectSingleNode("//rss/channel/title");
      if(titleNode != null)
        Console.WriteLine(titleNode.InnerText);
      Console.ReadKey();
    }
  }
}
```

ما از متد SelectSingleNode() برای یافتن المنت <title> استفاده کردیم. که این متد Xpath query را به عنوان یک پارامتر string در ورودی دریافت می کند. سپس چک می کنیم آیا متد فوق نتیجه ای را بازگردانده یا خیر. که اگر بازگردانده باشد، محتوای Inner Text عنصر node مورد نظر را در خروجی چاپ می کند که همان عنوان یا title فید RSS می باشد.

در مثال بعدی، از متد Select Nodes() برای یافتن کلیه آیتم های node در فید RSS و سپس چاپ اطلاعات درباره آن ها در خروجی استفاده کرده ایم :

```
using System;
```

```
using System.Text;
using System.Xml;

namespace ParsingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlDocument xmlDoc = new XmlDocument();
      xmlDoc.Load("rss.cnn.com/rss/edition_world.rss");
      XmlNodeList itemNodes = xmlDoc.SelectNodes("//rss/channel/item");
      foreach(XmlNode itemNode in itemNodes)
      {
        XmlNode titleNode = itemNode.SelectSingleNode("title");
        XmlNode dateNode = itemNode.SelectSingleNode("pubDate");
        if((titleNode != null) && (dateNode != null))
          Console.WriteLine(dateNode.InnerText + ": " + titleNode.InnerText);
      }
      Console.ReadKey();
    }
  }
}
```

کد ()Select Nodes همانند مثال قبلی، یک Xpath query را به عنوان پارامتر string ورودی دریافت کرده و سپس لیستی از اشیای XMLNode objects را به عنوان مجموعه ای از XMLNode List Collection بر می گرداند.

ما به وسیله یک حلقه foreach loop کد اجرای متد فوق را تکرار کرده و به ازای هر یک از آیتم های node، نام child node که همان title بوده و مقدار PubDate که اشاره به تاریخ نشر خبر دارد، را با استفاده از متد ()SelectSingleNode به صورت مستقیم از عنصر استخراج می کنیم. اگر هر دو آیتم فوق را با موفقیت دریافت کنیم، هر دو تاریخ و عنوان (title) را بر روی خروجی در یک خط چاپ کرده و به سراغ عنصر بعدی می رویم.

در مثال فوق، ما از هرکدام از آیتم node ها دو مقدار (value) متفاوت را درخواست می کردیم. به خاطر همین بود که هر item node را به صورت تکی مورد بررسی قرار داده و آن را پردازش می کنیم. به هر حال، اگر ما فقط یکی از مقادیر فوق مثل title را بخواهیم، می توانیم Xpath query فوق را به صورت زیر نیز بنویسیم :

```
//rss/channel/item/title
```

کد فوق نحوه انجام عملیات را به همراه چند دستور C# نشان می دهد :

```
//rss/channel/itXmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("rss.cnn.com/rss/edition_world.rss");
XmlNodeList titleNodes = xmlDoc.SelectNodes("//rss/channel/item/title");
foreach(XmlNode titleNode in titleNodes)
  Console.WriteLine(titleNode.InnerText);
Console.ReadKey();em/title
```

درس ۳۶ : آموزش نوشتن XML با استفاده از کلاس XmlWriter در زبان C#

آموزش نوشتن XML با استفاده از کلاس XmlWriter در زبان C# :

در درس های قبلی، به آموزش نحوه خواندن فایل های XML پرداختیم، اما در این درس و درس های بعدی قصد داریم تا بر روی نوشتن فایل های XML، کار کنیم. از آنجایی که زبان XML از متن ساده تشکیل شده است، شما می توانید به سادگی تگ های XML را درون یک فایل نوشته و سپس پسوند یا extensim با مقدار XML را به آن بدهید، اما مطمئن تر است که انجام این کار را به دست چهارچوب کاری .NET بسپارید.

همانند خواندن فایل های XML، نوشتن این فایل ها هم از طریق دو روش زیر ممکن است :

- نوشتن فایل XML با استفاده از XmlWriter
- نوشتن فایل XML با استفاده از XmlDocument

در این درس بر روی نوشتن فایل های XML با استفاده از XmlWriter تمرکز کرده و در درس بعدی به آموزش نحوه کار با XmlDocument خواهیم پرداخت.

تفاوت بین دو روش فوق در نوشتن فایل های XML، مربوط به میزان استفاده از حافظه (memory consuming) می باشد. XmlWriter به مراتب حافظه کمتری از XmlDocument استفاده کرده و این مسئله در هنگام نوشتن فایل های بزرگ XML، تفاوت زیادی را ایجاد خواهد کرد. تفاوت مهم دیگر این است که با استفاده از XmlDocument، شما می توانید همزمان یک فایل XML موجود را خوانده، آن را تغییر داده و سپس تغییرات ایجاد شده را بازنویسی کنید. اما در هنگام استفاده از XmlWriter، بایستی کل فایل XML را یکجا از روی چرک نویس بر روی فایل اصلی بنویسید. البته مورد دوم یک مشکل عمده نیست، اما انتخاب هر یک از روش های فوق بستگی به کار شما و ترجیحات شخصی دارد.

در مثال عملی زیر، کد نوشتن فایل XML با استفاده از کلاس XmlWriter قرار داده شده است :

```
using System;
using System.Text;
using System.Xml;
```

```
namespace WritingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlWriter xmlWriter = XmlWriter.Create("test.xml");

      xmlWriter.WriteStartDocument();
      xmlWriter.WriteStartElement("users");

      xmlWriter.WriteStartElement("user");
      xmlWriter.WriteAttributeString("age", "42");
      xmlWriter.WriteString("John Doe");
      xmlWriter.WriteEndElement();

      xmlWriter.WriteStartElement("user");
      xmlWriter.WriteAttributeString("age", "39");
      xmlWriter.WriteString("Jane Doe");

      xmlWriter.WriteEndDocument();
      xmlWriter.Close();
    }
  }
```

```
}
```

کد فوق، فایل XML زیر را تولید خواهد کرد :

```
<users>
 <user age="42">John Doe</user>
 <user age="39">Jane Doe</user>
</users>
```

در کد مثال فوق، ما با ایجاد یک نسخه جدید از کلاس XmlWriter کار را آغاز کرده ایم. کلاس فوق حداقل یک پارامتر را به عنوان ورودی دریافت می کند که ادرس محل ذخیره شدن فایل XML می باشد، اما پارامترهای دیگری را نیز می توان برای اهداف دیگر ارسال نمود. کار بعدی که بایستی انجام دهیم، فراخوانی متد Write Start Document() است. پس از آن، المنت آغازین با نام "user" را می نویسیم، کلاس XmlWriter Class متن فوق را به صورت <users> تغییر می دهد. قبل از بستن تگ فوق، یک تگ جدید با مقدار "user" را به عنوان فرزند یا child تگ user ایجاد خواهیم کرد. در مراحل بعدی به ترتیب با استفاده از متد WriteAttribute String()، یک خاصیت یا attribute را با نام age به المنت مورد نظر اضافه می کنیم. همچنین با استفاده از متد WriteString()، متن درونی یا inner text را برای تگ تولید خواهیم کرد. در پایان هم با فراخوانی متد WriteEndElement متد شده ایم که تگ المنت "user" را در انتها بسته ایم.

پروسه فوق می تواند برای اضافه کردن المنت های "user" بعدی نیز تکرار شود، به جز این که متد WriteEndElement() را مثل سری اول فراخوانی نمی کنیم. درواقع، این متد بایستی دو بار فراخوانی شود، زیرا دارای المنت باز "user" هستیم، اما کار فوق را کلاس XmlWriter در هنگام بستن فایل با فراخوانی متد WriteEndElement() انجام می دهد.

برای نوشتن اطلاعات فایل XML بر روی هارد دیسک، متد Close() فراخوانی خواهد شد. پس از فراخوانی این متد، می توانید فایل test.xml را باز کنید، این فایل درون پوشه ای خواهد بود که فایل EXE پروژه تان قرار داد و معمولا پوشه bin/debug است.

درس ۳۷ : آموزش نوشتن فایل XML با استفاده از کلاس XmlDocument در زبان C#

آموزش نوشتن فایل XML با استفاده از کلاس XmlDocument در زبان C# :

در درس قبلی، یک فایل XML را با استفاده از کلاس XmlWriter نوشتیم. اما در برخی از موارد، به خصوص ویرایش فایل های XML موجود، استفاده از کلاس XmlDocument کارآمدتر خواهد بود.

نکته : در هنگام نوشتن فایل های XML با استفاده از کلاس XmlDocument بایستی به استفاده زیاد این کلاس از حافظه توجه کنید، به خصوص در هنگام کار با فایل های بزرگ که ممکن است باعث مشکل در اجرای برنامه شود.

کد زیر، مثال عملی از نوشتن فایل های xml با استفاده از کلاس XmlDocument را نشان می دهد.

```
using System;
using System.Text;
using System.Xml;
using System.Xml.Serialization;

namespace WritingXml
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlDocument xmlDoc = new XmlDocument();
      XmlNode rootNode = xmlDoc.CreateElement("users");
      xmlDoc.AppendChild(rootNode);

      XmlNode userNode = xmlDoc.CreateElement("user");
```

```
        XmlAttribute attribute = xmlDoc.CreateAttribute("age");
        attribute.Value = "42";
        userNode.Attributes.Append(attribute);
        userNode.InnerText = "John Doe";
        rootNode.AppendChild(userNode);

        userNode = xmlDoc.CreateElement("user");
        attribute = xmlDoc.CreateAttribute("age");
        attribute.Value = "39";
        userNode.Attributes.Append(attribute);
        userNode.InnerText = "Jane Doe";
        rootNode.AppendChild(userNode);

        xmlDoc.Save("test-doc.xml");
    }
  }
}
```

کد فوق، فایل xml زیر را تولید خواهد کرد :

```
<users>
 <user age="42">John Doe</user>
 <user age="39">Jane Doe</user>
</users>
```

همانطور که متوجه شدید، روش نوشتن فایل های XML با استفاده از کلاس XmlWriter کمی سنتی گرا تر (object oriented) بوده و نیازمند کدنویسی بیشتری است، کاری که ما انجام می دهیم، تولید یک نسخه جدید از شی XmlDocument Object است که المنت های جدید و خواص یا attributes را با استفتده از متدهای ()CreateElement و CreateAttribute ایجاد می کند. هر زمانی که متدهای فوق فراخوانی می شوند، المنت جدید را یه به خود سند یا document اضافه کرده و یه به المنت زیری اضافه می شود. در مثال فوق، المنت "users" به صورت مستقیم به document اضافه شده، در حالی که المنت "user" به المنت اصلی یا rout element سند اضافه می شود. خواص یا attributes هم به المنت های مرتبط، توسط متد ()Append متعلق به خاصیت Attribute property اضافه می شوند. کل فایل XML نیز در انتهای کد و توسط متد Save () بر روی دیسک ذخیره خواهد شد.

همانطور که قبلا هم اشاره کردیم، ایجاد فایل XML کلاس XmlDocument نیازمند نوشتن کد مشتری است. اما حالتی را در نظر بگیرید که نیاز دارید، وارد یک فایل XML شده و چندین مقدار را می خواهید تغییر دهید. در صورت استفاده از کلاس XmlWriter، مجبور هستید ابتدا کل اطلاعات فایل XML را با استفاده از XmlReader خوانده، بر روی دیسک نگهداری کرده، سپس آن را تغییر دهید. در نهایت کل فایل آپدیت شده XML را به صورت یکجا، مجددا بازنویسی کنید.

اما به دلیل این که کلاس XmlDocument همه چیز را در حافظه نگهداری می کند، به روز رسانی فایل های XML بسیار ساده تر خواهد بود. کد مثال عملی زیر، فایل "text-doc-xml" که در درس قبلی ایجاد نمودیم را باز کرده، و خاصیت age کلیه تگ های user آن را یک واحد افزایش می دهد. به نحوه کار دقت کنید :

```
using System;
using System.Text;
using System.Xml;
using System.Xml.Serialization;

namespace WritingXml
```

```
{
  class Program
  {
    static void Main(string[] args)
    {
      XmlDocument xmlDoc = new XmlDocument();
      xmlDoc.Load("test-doc.xml");
      XmlNodeList userNodes = xmlDoc.SelectNodes("//users/user");
      foreach(XmlNode userNode in userNodes)
      {
        int age = int.Parse(userNode.Attributes["age"].Value);
        userNode.Attributes["age"].Value = (age + 1).ToString();
      }
      xmlDoc.Save("test-doc.xml");
    }
  }
}
```

در کد مثال فوق، ابتدا فایل XML را لود کرده و به دنبال المنت های <user> می گردیم. به صورت حلقه و تکرارگونه، هریک از مقادیر خاصیت های age را خوانده و در یک متغیر integer قرار می دهیم. سپس بعد از اضافه کردن یک واحد به خاصیت age، آن را مجددا در فایل بازنویسی می کنیم. در پایان نیز، کل سند XML را در درون همان فایل قبلی save کرده و اگر آن را مجددا باز کنید، خواهید دید سن هریک از کاربران یکسان بیشتر شده است.

درس ۳۸ : آموزش امکانات جدید در C# 3.0

آموزش امکانات جدید در زبان C# 3.0 :

همزمان با ارائه نسخه 3.5 چهارچوب کاری .NET و ویژوال استودیو 2008، مجموعه ای از قابلیت های جدید تحت نام مجموعه کد "Orcas" ، به نسخه C# 3.0 اضافه شده است. در درس های این بخش، قصد داریم تا با مهم ترین ویژگی های جدید اضافه شده که تلاشی جهت بالا بردن قابلیت کدنویسی کاربران است آشنا شویم.

نکته مهم : توجه داشته باشید که برای استفاده از ویژگی های جدید نسخه C# 3.0، بایستی حداقل چهارچوب کاری .NET 3.5 و نرم افزار ویژوال استودیو 2008 بر روی سیستم شما نصب شده باشد.

در صورت داشتن نرم افزارهایی با ورژن پایین تر از موارد ذکر شده، کامپایل و اجرای کدهای جدید با خطا مواجه خواهد شد.

درس ۳۹ : آموزش خواص اتوماتیک Automatic Properties در C#

آموزش خواص اتوماتیک Automatic Properties در زبان C# :

یکی از بزرگ ترین مشکلاتی که برنامه نویسان برای نوشتن کدهای شی گرا یا object oriented داشتند، امکان تعیین خواص عمومی(Public Properties) برای فیلدهای خصوصی (private fields) بود. این کار کمی خسته کننده بود، زیرا تغییر با تمامی خواص به صورت یک متد get و set ساده با نگاشت به فیلد خصوصی تعریف می شد و هیچ هوشمندی خاصی در کار نبود. مثل کد زیر :

```
private string name;


public string Name
{
  get { return name; }
  set { name = value; }
}
```

در هنگام تعریف یک خاصیت یا Property ساده مثل کد فوق، می توان آن را به صورت عمومی public تعریف نموده و به صورت مستقیم برای متغیر به کار برد، بدون این که نیاز داشته باشد لایه ای مخصوص property ایجاد کنیم. اما ما بر طبق آموزش های برنامه نویسی شی گرا cop، بایستی کار را به صورت فوق انجام دهیم با این وجود برخی برنامه نویسان از به کار بردن روش ساده فوق، خودداری می کنند. اما با آمدن نسخه C# 3.0، دیگر بر این دو راهی قرار نگرفته و کد مثال فوق را می توان به صورت زیر نوشت :

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

یا حتی به روش کم حجم تری مثل زیر نیز می توان نوشت :

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

در کد فوق، نیازی به تعریف field نداشته و کد اضافه برای متد get و set نیاز نیست. کلیه عملیات مورد نیاز به صورت خودکار توسط کامپایلر انجام می شود. خود کامپایلر C#، به صورت اتوماتیک یک فیلد خصوصی private field ایجاد کرده و متدهای get و set لازم را با کدهای مناسب جهت خواندن و نوشتن فیلد تولید می کند.

اگر به کد فوق از بیرون نگاه کنید، ظاهری شبیه یک property معمولی دارد، اما با همین مقدار کاهش کد، حجم تایپ شما بسیار کمتر شده و در کلاس خلاصه تر به نظر می رسد. به طور قطع می توانید همچنان از روش قدیمی نیز استفاده کنید، همانطور که در مثال درس نشان دادیم، اما روش دوم راه ساده تری است.

درس ۴۰ : آموزش مقداردهی اولیه اشیاء object initializer در C#

آموزش مقداردهی اولیه اشیاء object initializer در زبان C# :

در C# 3.0، هر دو کار مقداردهی اولیه یا initializing اشیاء (objects) و مجموعه ها (collections) بسیار ساده تر شده است و کلاس ساده Car Class را که در آن به وسیله خواص اتوماتیک، متغیرهای خود را همانند درس قبل مقداردهی کرده ایم را در نظر بگیرید :

```
class Car
{
   public string Name { get; set; }
   public Color Color { get; set; }
}
```

در زبان C# 2.0، ما مجبور بودیم برای ایجاد یک نمونه جدید از کلاس Car و تنظیم خصوصیات آن، کدی در حجم زیر را بنویسیم :

```
Car car = new Car();
car.Name = "Chevrolet Corvette";
car.Color = Color.Yellow;
```

کد فوق هم خوب است، ولی با ارائه C# 3.0 و به لطف ساختار مقداردهی object های جدید، می توان کد فوق را کمی مفهوم تر و تمیزتر به صورت زیر نوشت :

```
Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow };
```

همانطور که در کد فوق مشاهده می کنید، ما در مقابل نام Car از یک جفت براکت {} استفاده کرده و درون آن به تمامی خصوصیات عمومی کلاس Car Class دسترسی داشته و آن ها را مقداردهی کرده ایم. روش زیر هم کمی حجم تایپ را کاهش داده و هم فضای کدنویسی را. نکته جالب در روش فوق این است که آن را می توان تو در تو نیز نوشت.

کد مثال زیر را که در آن چندین خاصیت را برای کلاس Car تعریف کرده ایم را در نظر بگیرید.

```
class Car
{
  public string Name { get; set; }
  public Color Color { get; set; }
  public CarManufacturer Manufacturer { get; set; }
}

class CarManufacturer
{
  public string Name { get; set; }
  public string Country { get; set; }
}
```

برای مقداردهی یک شی جدید در C# 2.0، مجبوریم کدی به صورت زیر بنویسیم :

```
Car car = new Car();
car.Name = "Corvette";
car.Color = Color.Yellow;
car.Manufacturer = new CarManufacturer();
car.Manufacturer.Name = "Chevrolet";
car.Manufacturer.Country = "USA";
```

اما در C# به صورت زیر نوشته می شود :

```
Car car = new Car {
        Name = "Chevrolet Corvette",
        Color = Color.Yellow,
        Manufacturer = new CarManufacturer {
          Name = "Chevrolet",
          Country = "USA"
        }
      };
```

یا حتی می توانید برای کاهش حجم فایل، در صورتی که به خوانایی کد اهمیت نمی دهید، می توان به صورت زیر نیز نوشت :

```
Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow, Manufacturer = new CarManufacturer { Name = "Chevrolet", Country = "USA" } };
```

نکته : همانند تعریف خصوصیات اتوماتیک (outomatic properties) که در درس قبل نشان دادیم، روش فوق یک جایگزین جدید است و کماکان می توان از روش های قدیم هم استفاده کرد.

درس ۴۱ : آموزش مقداردهی مجموعه ها Collection در C#

آموزش مقداردهی مجموعه ها Collection در زبان C# :

همانطور که C# 3.0 یک روش جدید را برای مقداردهی اشیاء objects ارائه داده است، ساختار دستوری جدیدی نیز برای مقداردهی اولیه list ها با مجموعه ای از آیتم ها، در دسترس است. برای این منظور از کلاس Car Class را که در درس قبل نیز نشان دادیم، استفاده می کنیم :

```
class Car
```

```
{
   public string Name { get; set; }
   public Color Color { get; set; }
}
```

اگر می خواستیم یک لیست حاوی تعدای Car را در C# 2.0 تعریف کنیم، بایستی از کدی مثل مثال زیر استفاده می کردیم :

```
Car car;
List<car> cars = new List<car>();

car = new Car();
car.Name = "Corvette";
car.Color = Color.Yellow;
cars.Add(car);

car = new Car();
car.Name = "Golf";
car.Color = Color.Blue;
cars.Add(car);
        </car></car>
```

با استفاده از روش جدید مقداردهی اشیا object، کد فوق را می توان به صورت زیر بازنویسی کرد :

```
List<car> cars = new List<car>();
cars.Add(new Car { Name = "Corvette", Color = Color.Yellow });
cars.Add(new Car { Name = "Golf", Color = Color.Blue});
```

```
</car></car>
```

کد فوق را حتی با ویژگی مقداردهی جدید object ها می توان به صورت زیر خلاصه تر کرد :

```
List<car> cars = new List<car>
{
  new Car { Name = "Corvette", Color = Color.Yellow },
  new Car { Name = "Golf", Color = Color.Blue}
};
```

```
</car></car>
```

نوشتن کد فوق به صورت تک خطی هم ممکن است :

```
List<car> cars = new List<car> { new Car { Name = "Corvette", Color = Color.Yellow }, new Car {
Name = "Golf", Color = Color.Blue} };
```

```
</car></car>
```

با روش فوق، ۱۰ خط فقط به یک خط (البته کمی طولانی) تبدیل شده و این به دلیل ویژگی جدید مقداردهی objects هاست.

درس ۴۲ : آموزش متدهای توسعه یافته Extensim Methods در C#

آموزش متدهای توسعه یافته Extensim Methods در زبان C# :

قابلیت جدید دیگر زبان C# 3.0، متدهای توسعه یافته یا Extensim Methods است. این ویژگی به شما امکان می دهد تا قابلیت های جدیدی را به یک نوع داده ای یا type موجود اضافه کنید، بدون این که نیاز داشته باشید آن type را به صورت یک زیرمجموعه یا کلاس به ارث رفته درآورد یا مجددا کامپایلش کنید. به

عبارت دیگر می توانید عملکردهای جدیدی را به یک نوع داده ای اضافه کنید، بدون این که نسخه اصلی آن را تغییر دهید.

برای مثال ممکن است در موردی بخواهید بدانید یک نوع داده ای متنی یا string شامل اعداد است یا خیر. راه حل عادی برای انجام این کار، تعریف یک تابع یا function بوده و هر زمان که نیاز دارید آن تابع را فراخوانی می کنید. پس از این که چند مدل از این نوع تابع ها را تعریف کردید، می توانید آن ها را در یک کلاس به صورت زیر قرار دهید (utility class) :

```
public class MyUtils
{
   public static bool IsNumeric(string s)
   {
      float output;
      return float.TryParse(s, out output);
   }
}
```

پس می توانید یک متغیر متنی یا string را به صورت زیر چک نمایید :

```
string test = "4";
if (MyUtils.IsNumeric(test))
   Console.WriteLine("Yes");
else
   Console.WriteLine("No");
```

اما به وسیله متدهای توسعه یافته یا Extension Methods، می توانید کلاس string را به گونه ای گسترش داده تا به صورت مستقیم عملیات فوق را انجام دهد. این کار با تعیین یک کلاس ثابت Static Class و تعریف

چندین تابع ثابت Static Methods درون آن به عنوان کتابخانه ای از کدها، صورت می گیرد. مثال زیر، نحوه کدنویسی را به صورت عملی نشان داده است :

```
public static class MyExtensionMethods
{
  public static bool IsNumeric(this string s)
  {
    float output;
    return float.TryParse(s, out output);
  }
}
```

نکته : تنها چیزی که این متد را از سایر Static Methods ها، متمایز می کند، به کار بردن واژه کلیدی this در بخش پرانتز پارامترهای تابع است. این کلمه کلیدی به کامپایلر اطلاع می دهد که این تابع یک Extension Methods می باشد و تنها کار لازم جهت تبدیل یک متد معمولی به یک متد توسعه یافته است.

پس از انجام مراحل فوق، می توانید تابع IsNumeric() را به صورت مستقیم بر روی شی string به کار برید ، به صورت کد زیر :

```
string test = "4";
if (test.IsNumeric())
  Console.WriteLine("Yes");
else
  Console.WriteLine("No");
```

درس ۴۳ : آموزش خواندن و نوشتن فایل ها در C#

آموزش خواندن و نوشتن فایل ها در زبان C# :

در این بخش ، قصد داریم تا نحوه خواندن (Reading) و نوشتن (Writing) در فایل ها، به وسیله زبان C# را آموزش دهیم. خوشبختانه، زبان C# با امکاناتی که ارائه داده است، انجام این امور را بسیار ساده کرده است. کلاس File Class از System IO namespace، تقریبا شامل تمام چیزهایی است که برای خواندن و نوشتن فایل ها در C# نیاز داریم.

در اولین مثال این درس، یک ویرایشگر متنی ساده را کدنویسی می کنیم. این ویرایشگر به حدی ساده است که به وسیله آن فقط می توان محتویات یک فایل را خوانده و سپس در هر زمان یک جدید خط به آن اضافه نمود. اما در عین حال، بسیاری از ویژگی های کار با کلاس Field Class را نشان می دهد. به صورت کد زیر :

```
using System;
using System.IO;

namespace FileHandlingArticleApp
{
  class Program
  {
    static void Main(string[] args)
    {
      if(File.Exists("test.txt"))
      {
        string content = File.ReadAllText("test.txt");
        Console.WriteLine("Current content of file:");
        Console.WriteLine(content);
      }
```

```
        Console.WriteLine("Please enter new content for the file:");
        string newContent = Console.ReadLine();
        File.WriteAllText("test.txt", newContent);
    }
  }
}
```

اگر دقت کرده باشید در کد مثال فوق، کلاس File Class را در سه نقطه استفاده کرده ایم :

- اول این که از آن استفاده کرده این تا ببینیم آیا فایل متنی مورد نظر وجود دارد یا خیر، file exist.
- از متد ReadAllText() برای خواند کل محتویات فایل استفاده شده است.
- در انتها هم از متد WriteAllText() برای اضافه کردن خط محتویات جدید به فایل استفاده کرده ایم.

باز هم اگر دقت کرده باشید، از یک آدرس مطلق برای فراخوانی فایل متنی مورد نظر خود استفاده نکرده و فقط نام فایل را به کار برده ایم. انجام این باعث قرار گرفتن فایل متنی در همان پوشه فایل اجرایی برنامه می شود که برای این مرحله مناسب است.

از طرف دیگر، مثال عملی ارائه شده بسیار ساده جهت یادگیری است : ما چک کردیم فایل متنی وجود دارد یا خیر، سپس کل مجتویات آن را در Cousole نشان دادیم. در مرحله دوم از کاربر خواسته ایم تا متد مورد نظر خود را تایپ نموده و بعد محتویات جدید را به فایل اضافه نمودیم. البته این کار باعث بازنویسی محتویات فایل قبلی می شود و درواقع آن را آپدیت می کند، که برای مرحله اول آموزش خوب است.

نکته کاربردی : می توان به جای متد WriteAllText() از متد Append AllText() به صورت مثال زیر استفاده کنیم تا محتویات جدید را به انتهای فایل موجود اضافه کرده و باعث از بین رفتن محتوی اولیه فایل نشود.

```
File.AppendAllText("test.txt", newContent);
```

اگر کد فوق را در مثال اول تغییر دهید، خواهید دید که برنامه به جای رونویسی کامل فایل اولیه، متن را به انتهای آن اضافه می کند. اما هنوز ما فقط یک خط متن می توانیم به برنامه بدهیم. بیایید مثال قبل را کمی گسترش داده و آن را هوشمندتر کنیم. خط های کد زیر را با خط های انتهای مثال اول جایگزین کنید :

```
Console.WriteLine("Please enter new content for the file - type exit and press enter to finish
editing:");
string newContent = Console.ReadLine();
while(newContent != "exit")
{
   File.AppendAllText("test.txt", newContent + Environment.NewLine);
   newContent = Console.ReadLine();
```

همانطور که مشاهده می کنید، ما کاربر را راهنمایی کردیم که در صورت تمایل برای اتمام عمل اضافه کردن متن به فایل موجود، عبارت "exit" را تایپ کند. در صورت عدم وارد نمودن این واژه، کاربر می تواند هر تعداد خط متن جدید که می خواهد به فایل اضافه کرده و برنامه با ارائه متد درخواست متن، همچنان ورودی دریافت می کند. ما همچنین یک کد جدید را به انتهای دستور newline به نام Enviroment.NewLine اضافه کرده ایم تا محیط برنامه کمی شبیه ادیتورهای متنی شود.

از طرف دیگر، به جای این که در هر بار اجرای کدهای برنامه، محتویات به فایل اضافه کنیم، به کاربران راه حل زیر می تواند کمی بهتر باشد :

```
Console.WriteLine("Please enter new content for the file - type exit and press enter to finish
editing:");
using(StreamWriter sw = new StreamWriter("test.txt"))
{
   string newContent = Console.ReadLine();
```

```
  while(newContent != "exit")
  {
    sw.Write(newContent + Environment.NewLine);
    newContent = Console.ReadLine();
  }
}
```

استفاده از قابلیت Steams کمی فراتر از آموزش های این بخش است، اما خلاصه کاربرد آن این است که به وسیله آن، فایل متنی مورد نظر را یک بار باز کرده و هرچقدر محتویات داریم، به آن اضافه کرده و در پایان کار به یک باره فایل را می بندیم.

از طرف دیگر، در مثال فوق ما از قابلیت دستور Using() در C# استفاده کرده ایم. دستور Using() این اطمینان را به ما می دهد که ارجاع به فایل به محض این که برنامه از Scope کد خارج شد، که در اینجا انتهای براکت ها {} است، به صورت خودکار بسته می شود. در صورت عدم استفاده از using() بایستی در انتهای نمونه شی StreamWriter، جهت بستن فایل متنی، متد Close() را فراخوانی کنید.

درس ۴۴ : آموزش کار با فایل ها و پوشه ها در زبان C#

آموزش کار با فایل ها و پوشه ها در زبان C# :

در درس قبلی، به آموزش نحوه خواندن و نوشتن در فایل های ساده متنی توسط C# پرداختیم. ما برای آموزش از کلاس File Class برای اولین بار استفاده کردیم، اما این کلاس قابلیت های بسیار بیشتری از خواندن و نوشتن فایل های متنی دارد.

وقتی کلاس File Class را با کلاس Directory Class ترکیب کنید، می توانید بسیاری از کارهای فایل سیستمی مثل تغییر نام فایل ها و پوشه، تغییر مکان آن ها و حتی حذف و ... را انجام دهید.

در این بخش چندین مثال مختلف را برای انجام امور فایل ها نشان خواهیم داد. از آنجایی که کد این مثال ها بسیار ساده و راحت برای استفاده است، از ارائه توضیحات بیشتر خودداری شده است. شما در هنگام کار با فایل ها و پوشه ها، فقط بایستی از دو چیز اطمینان حاصل کنید :

اضافه کردن مجموعه کلاس های Sysytm.IO namespace به ابتدای فایل ها، به صورت زیر :

```
using System.IO;
```

از طرف دیگر توجه داشته باشید که ما هیچ کار خاصی برای مدیریت خطاها یا exception در کدها انجام نداده ایم. البته در ابتدای هر مثال، وجود یا عدم وجود فایل مورد نظر جهت تغییر، چک می شود، اما در مواردی که مشکلاتی پیش بیاید، مدیریت خطا پیش بینی نشده است. در هنگام انجام عملیات های IO، مدیریت خطا یا exception یک مسئله مهم است و برای این منظور به بخش آموزش مدیریت خطاها در C# بروید.

در کلیه مثال های این درس، ما به صورت مستقیم از نام فایل ها و پوشه ها استفاده کرده و آدرس کامل ارائه نشده است. زیرا فرض بر این است که فایل EXE اجرایی برنامه C# در همان پوشه فایل می باشد. در بخش Project setting برنامه تان می توانید مشاهده کنید که فایل EXE در چه مسیری ایجاد می شود.

آموزش حذف یک فایل در زبان C# :

کد زیر نحوه حذف فایل ها در زبان C# را نشان داده است :

```
if(File.Exists("test.txt"))
{
  File.Delete("test.txt");
  if(File.Exists("test.txt") == false)
    Console.WriteLine("File deleted...");
}
else
```

```
  Console.WriteLine("File test.txt does not yet exist!");
Console.ReadKey();
```

آموزش حذف یک پوشه در زبان C# :

کد زیر نحوه حذف پوشه ها در زبان C# را نشان داده است :

```
if(Directory.Exists("testdir"))
{
  Directory.Delete("testdir");
  if(Directory.Exists("testdir") == false)
    Console.WriteLine("Directory deleted...");
}
else
  Console.WriteLine("Directory testdir does not yet exist!");
Console.ReadKey();
```

در کد مثال فوق، اگر پوشه testdir خالی نباشد، برنامه دچار خطا یا exeption می شود. چرا؟ به دلیل این که این نسخه از متد ()Delete در کلاس Directory Class ، فقط بر روی پوشه های خالی کار می کند. برای جلوگیری از خطا، می توان کد متد را به صورت زیر تغییر دهید :

```
Directory.Delete("testdir", true);
```

پارامتر اضافه شده به متد فوق، خاطر نشان می کند که متد ()Delete بازگشتی یا vecursive باشد. به این معنا که این متد ابتدا از پوشه های درون پوشه اصلی شروع کرده و با حذف آن ها، در انتها به سراغ پوشه اصلی می آید .

آموزش تغییر نام یک فایل در زبان C# :

کد زیر، نحوه تغییر نام یک فایل به وشیله C# را نشان می دهد :

```
if(File.Exists("test.txt"))
```

```
{
  Console.WriteLine("Please enter a new name for this file:");
  string newFilename = Console.ReadLine();
  if(newFilename != String.Empty)
  {
    File.Move("test.txt", newFilename);
    if(File.Exists(newFilename))
    {
      Console.WriteLine("The file was renamed to " + newFilename);
      Console.ReadKey();
    }
  }
}
```

اگر دقت کرده باشید، در کد مثال فوق، از تابع Move() برای تغییر نام فایل استفاده شده است. اما چرا از متد Rename() استفاده نکرده ایم، زیرا متد Move() درواقع همان کار متد Rename() را انجام می دهد.

آموزش تغییر نام یک پوشه در C# :

```
if(Directory.Exists("testdir"))
{
  Console.WriteLine("Please enter a new name for this directory:");
  string newDirName = Console.ReadLine();
  if(newDirName != String.Empty)
  {
    Directory.Move("testdir", newDirName);
    if(Directory.Exists(newDirName))
```

```
    {
      Console.WriteLine("The directory was renamed to " + newDirName);
      Console.ReadKey();
    }
  }
}
```

آموزش ایجاد یک پوشه جدید در زبان C# :

```
Console.WriteLine("Please enter a name for the new directory:");
string newDirName = Console.ReadLine();
if(newDirName != String.Empty)
{
  Directory.CreateDirectory(newDirName);
  if(Directory.Exists(newDirName))
  {
    Console.WriteLine("The directory was created!");
    Console.ReadKey();
  }
}
```

درس ۴۵ : آموزش استخراج اطلاعات فایل و پوشه ها در زبان C#

آموزش استخراج اطلاعات فایل و پوشه ها در زبان C# :

کلاس های File Class و Directory Class که در بخش های قبل استفاده کردیم، برای کار مستقیم با فایل ها و دستکاری پوشه ها، بسیار مناسب هستند. اما گاهی اوقات نیاز داریم تا اطلاعاتی را درباره یک فایل یا پوشه بدست آوریم، به جای این که لزوما آن را تغییر دهیم. در این گونه موارد نیز System.IO namespace

به کمک آمده و کلاس های FileInfo Class و DirectoryInfo Class را در اختیارمان قرار می دهد. در این درس قصد داریم تا با ارائه مثال هایی، نحوه کار با این کلاس ها را آموزش دهیم.

در کد مثال زیر، نحوه استفاده از کلاس FileInfo Class را نشان می دهد :

```
static void Main(string[] args)
{
  FileInfo fi = new FileInfo(System.Reflection.Assembly.GetExecutingAssembly().Location);
  if(fi != null)
    Console.WriteLine(String.Format("Information about file: {0}, {1} bytes, last modified on
{2} - Full path: {3}", fi.Name, fi.Length, fi.LastWriteTime, fi.FullName));
  Console.ReadKey();
}
```

در کد مثال فوق، ابتدا یک نمونه جدید از کلاس FileInfo Class را ایجاد کرده ایم. این نمونه یک پارامتر که حاوی آدرس فایلی که می خواهیم راجع به آن، اطلاعات کسب کنیم را دریافت می کند. می توان نام هر فایل دلخواهی را در این پارامتر قرار داده اما ما برای جالب تر کردن مثال، آدرس فایل اجرایی EXE ای که در حال کامپایل برنامه کنونی ماست را ارسال کردیم. از آنجایی که ما به فایل اجرایی یک پروژه از طریق Console Application دسترسی نداریم (این فایل بخشی از WinForms assembly است)، از یک Reflection برای دسترسی به آدرس assembly جاری استفاده کرده ایم. انجام این کار در چهارچوب مطالب آموزشی این درس نیست، اما برای زمان حال، آن را در نظر داشته باشید تا در بخش جداگانه ای به آموزش آن بپردازیم. تا زمانی که یک نسخه از کلاس FileInfo داشته باشیم، می توانیم اطلاعات مختلفی را راجع به فایل ارسال شده به آن، دریافت کنیم. پروژه کد مثال قبل را اجرا نموده و خواهدید دید که خیلی ساده و مرتب، کلاس FileInfo اطلاعات بسیار کاملی را راجع به فایل در اختیارمان قرار می دهد، حتی shortcut هایی که به متدهای کلاس File ارتباط دارند.

با استفاده از کلاس FileInfo، به اطلاعات یک فایل تنها دسترسی داشتیم. اما با استفاده از کلاس DirectoryInfo می توانیم اطلاعات مربوط به تمامی فایل ها و پوشه های موجود در یک پوشه را بدست آوریم، که امکان بسیار مناسبی است. مثال زیر، نحوه انجام کار را نشان می دهد :

```
 DirectoryInfo di = new
DirectoryInfo(Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().
Location));
if(di != null)
{
   FileInfo[] subFiles = di.GetFiles();
   if(subFiles.Length > 0)
   {
      Console.WriteLine("Files:");
      foreach(FileInfo subFile in subFiles)
      {
         Console.WriteLine("   " + subFile.Name + " (" + subFile.Length + " bytes)");
      }
   }
   Console.ReadKey();
}
```

در کد مثال فوق، به جای یک نسخه از کلاس FileInfo یک نسخه از کلاس DirectoryInfo را ایجاد کرده ایم. سپس از روشی مشابه مثال اول، برای دریافت مسیر فایل مورد نظر استفاده کرده و با کمک متد GetDirectoryName() از کلاس Path Class، آدرس پوشه را و نام آن را از مسیر استخراج کرده ایم. ما از

متد GetFiles() استفاده کرده ایم که هر عضو آرایه، نماینده یک فایل در پوشه است. سپس با کمک یک حلقه loop، درون آرایه حرکت کرده و نام فایل و سایز هر کدام را در خروجی چاپ کرده ایم.

همچنین می توانید لیست کامل پوشه های موجود در پوشه جاری را با استفاده از کد زیر به دست آورید :

```
DirectoryInfo[] subDirs = di.GetDirectories();
if(subDirs.Length > 0)
{
  Console.WriteLine("Directories:");
  foreach(DirectoryInfo subDir in subDirs)
  {
    Console.WriteLine("  " + subDir.Name);
  }
}
```

در برخی موارد، ممکن است شما فقط فایل ها یا پوشه هایی با نام یا پسوند خاص را بخواهید مشاهده کنید، خوشبختانه کلاس های FileInfo و DirectoryInfo پشتیبانی خوبی از این مسئله، انجام می دهند.

به کمک کد مثال زیر، می توانیم به کلیه فایل های دارای پسوند exe. و پوشه جاری دسترسی داشته باشیم :

```
FileInfo[] subFiles = di.GetFiles("*.exe");
```

همچنین کد زیر، لیست کامل نام پوشه هایی که در نام آن ها عبارت "test" وجود دارد را نشان می دهد :

```
DirectoryInfo[] subDirs = di.GetDirectories("*test*");
```

ما همچنین می توانیم به صورت بازگشتی به دنبال نام فایل ها و پوشه ها بگردیم. یعنی این که زیر پوشه ها و پوشه های تو در توی یک پوشه را کاملا جستجو کنیم، به کمک کد مثال زیر :

```
FileInfo[] subFiles = di.GetFiles("*.exe", SearchOption.AllDirectories);
```

اگر بخواهیم فقط اولین پوشه (در بالاترین رده) را جستجو کنیم، کد به صورت زیر تغییر می کند :

```
FileInfo[] subFiles = di.GetFiles("*.exe", SearchOption.TopDirectoryOnly);
```

درس ۴۶ : آموزش Reflection در زبان C#

آموزش Reflection در زبان C# :

در عمل کامپیوتر، مفهوم reflection به صورت زیر توضیح داده شده است :

"reflection به وسیله ای گفته می شود که در آن، برنامه کامپیوتری می تواند رفتار خود را مشاهده کرده و در صورت نیاز ساختار و عملکرد خود را تغییر دهید."

تعریف فوق، حالتی است که دقیقا C# مطابق آن عمل می کند. البته اگر بتوانید این موضوع را در کدنویسی خود به خوبی درک کنید، قابلیت امتحان کردن و تغییر اطلاعات برنامه تان در هنگام اجرا، پتانسیل و نیروی فوق العاده ای را در اختیار شما قرار می دهد. استفاده از Reflection به معنای واقعی کلمه، در زبان C#، بسیار ساده و کاربردی است.

در درس های این بخش، قصد داریم تا به آموزش کامل مفهوم Reflection در زبان C# و بررسی زوایای مختلف آن، همراه با مثال های عملی و وسورس کد پرداخته، تا متوجه شوید استفاده از این قابلیت، چقدر مفید و کاربردی است.

برای شروع آموزش و جالب تر شدن مسئله، به ارائه یک مثال عملی در زمینه reflection می پردازیم. قابلیت Reflection، به راحتی این سوال را که برنامه نویسان حرفه ای و تازه وارد، مدام با آن رو به رو هستند را پاسخ می دهد. چگونه می توانیم در زمان اجرا یا Run Time، مقدار یک متغیر را با داشتن نام آن تغییر دهیم؟

در کد مثال عملی زیر، با دقت بنگرید. سپس در ادامه این درس و بخش های بعدی به تشریح تکنیک های به کار رفته خواهیم پرداخت :

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
  class Program
  {
    private static int a = 5, b = 10, c = 20;

    static void Main(string[] args)
    {
      Console.WriteLine("a + b + c = " + (a + b + c));
      Console.WriteLine("Please enter the name of the variable that you wish to change:");
      string varName = Console.ReadLine();
      Type t = typeof(Program);
      FieldInfo fieldInfo = t.GetField(varName, BindingFlags.NonPublic | BindingFlags.Static);
      if(fieldInfo != null)
      {
        Console.WriteLine("The current value of " + fieldInfo.Name + " is " +
fieldInfo.GetValue(null) + ". You may enter a new value now:");
        string newValue = Console.ReadLine();
        int newInt;
        if(int.TryParse(newValue, out newInt))
```

```
        {
          fieldInfo.SetValue(null, newInt);
          Console.WriteLine("a + b + c = " + (a + b + c));
        }
        Console.ReadKey();
      }
    }
  }
}
```

کد مثال فوق را اجرا کرده و مشاهده نمایید چگونه کار می کند. در درس های بعدی به تشریح نحوه عملکرد کد فوق خواهیم پرداخت.

درس ۴۷ : آموزش right type در Reflection زبان C#

آموزش right type در Reflection زبان C# :

کلاس Type Class اساس و پایه مفهوم Reflecton در زبان C# است. کلاس Type به عنوان اطلاعات لازم اجرا، برای یک اسمبلی، ماژول و یا type استفاده می شود.

خوشبختانه، به دلیل این که همه کلاس ها در C# از کلاس object class به ارث رفته اند، ایجاد یک رفرنس به نوع (Type) یک شی (object) ، با استفاده از متد GetType() بسیار ساده است. اگر همچنین، اطلاعاتی در رابط یک نوع (Type) مقداردهی نشده (non-instantiated) بخواهید کسب کنید، می توانید از متد عمومی (Global) موجود به نام typeof() استفاده کنید، که همان کار GetType() را انجام می دهد.

در مثال عملی زیر، ما از هر دو روش برای اجرای کد خود استفاده کرده ایم :

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

```
using System.Reflection;


namespace ReflectionTest
{
  class Program
  {
    static void Main(string[] args)
    {
      string test = "test";
      Console.WriteLine(test.GetType().FullName);
      Console.WriteLine(typeof(Int32).FullName);
      Console.ReadKey();
    }
  }
}
```

ما در مثال فوق از متد GetType() برای عمل برای دو متغیر مورد نظر خود و از متد typeof() برای کار با کلاس شناخته شده Int32 استفاده کرده ایم. همان طور که مشاهده خواهید کرد، خروجی اجرای هر دو متد اشاره شده، یک TypeObject است، که به راحتی می توان مقدار خاصیت FullName Property آن را خواند.

در برخی موارد، شما مکن است فقط نام نوع یا type ای که به دنبال آن هستید، را بدانید. در چنین موردی، بایستی یک رفرنس را از طریق یک اسمبلی مناسب به type مورد نظر ایجاد نمایید. در مثال بعدی، ما یک رفرنس به اسمبلی اجرایی، کد اسمبلی اجرا کننده کد جاری نیز است، ایجاد کرده و سپس لیست کلیه type های موجود در آن را استخراج کرده ایم.

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

```
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
  class Program
  {
    static void Main(string[] args)
    {
      Assembly assembly = Assembly.GetExecutingAssembly();
      Type[] assemblyTypes = assembly.GetTypes();
      foreach(Type t in assemblyTypes)
        Console.WriteLine(t.Name);
      Console.ReadKey();
    }
  }

  class DummyClass
  {
    //Just here to make the output a tad less boring :)
  }
}
```

خروجی کد مثال فوق، نام دو کلاس تعریف شده در اسمبلی یعنی کلاس Program و کلاس DumingClass است. اما در پروژه های بزرگتر، خروجی های لیست کامل تمامی کلاس های پروژه خواهد بود.

در این مثال، ما فقط نام Type ها را استخراج کردیم، اما به دلیل امکان ایجاد رفرنس به Type ها، کارهای بسیاری می توان با نام آن ها انجام داد. در درس بعدی، به تشریح و آموزش کامل کاربرد این رفرنس ها خواهیم پرداخت.

درس ۴۸ : آموزش نمونه سازی کلاس ها در زبان C#

آموزش نمونه سازی کلاس Class در زبان C# :

تا کنون در چهارچوب کاری .NET. با نمونه ها types و اشیای object ای که از قبل نمونه سازی شده اند، کار کرده ایم. اما به وسیله قابلیت Reflection، می خواهیم عمل نمونه سازی را در هنگام اجرا یا RunTime انجام دهیم و برای این منظور داشتن نام کلاسی که می خواهیم از آن نمونه بسازیم، ضروری است. چندین راه برای انجام کار فوق وجود دارد. اما من روشی را ترجیح می دهد که در آن یک رفرنس به تابع سازنده شی یا Constructor ایجاد کرده و آن را فعال می کنیم. سپس مقدار برگشتی از تابع را به عنوان نمونه خود به کار می بریم.

کد زیر، به صورت عملی نحوه نمونه سازی از کلاس ها در زمان اجرای برنامه های C# را نشان داده است. ابتدا کد مثال را مرور کرده و در ادامه به تشریح آن خواهیم پرداخت :

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
  class Program
  {
    static void Main(string[] args)
```

```
    {
      Type testType = typeof(TestClass);
      ConstructorInfo ctor = testType.GetConstructor(System.Type.EmptyTypes);
      if(ctor != null)
      {
        object instance = ctor.Invoke(null);
        MethodInfo methodInfo = testType.GetMethod("TestMethod");
        Console.WriteLine(methodInfo.Invoke(instance, new object[] { 10 }));
      }
      Console.ReadKey();
    }
  }

  public class TestClass
  {
    private int testValue = 42;

    public int TestMethod(int numberToAdd)
    {
      return this.testValue + numberToAdd;
    }
  }
}
```

در کد مثال فوق، یک کلاس ساده به نام TextClass را جهت نمایش نحوه کارایی ایجاد کرده ایم. این کلاس دارای یک فیلد خصوصی (Private field) و یک متد عمومی (public method) می باشد. هر کلاس، مقدار

فیلد خصوصی را به همراه مقدار پارامتر ارسالی که به آن اضافه شده است را بر می گرداند. حال کاری که ما می خواهیم انجام دهیم، ایجاد یک نمونه جدید از کلاس TextClass و سپس فراخوانی متد TextMethod و ارسال خروجی بر روی Console است.

همچنین در کد مثال فوق، ما امکان استفاده مستقیم از متد ()typeof بر روی کلاس TextClass را داشتیم. اما در برخی موارد، شما ممکن است مجبور شوید منحصرا از نام خود کلاس مورد نظر استفاده کنید. در چنین موردی، بایستی یک رفرنس را از طریق اسمبلی که کلاس در آن تعریف شده است، ایجاد نمایید، همانند آنچه در بخش کار با Type خدمتتان عرض کردیم.

بنابراین با داشتن یک رفرنس Type به کلاس مورد نظر، می توانیم تابع سازنده پیش فرض default Constructor را با استفاده از قید ()GetConstructor احضار کنیم. سپس مقدار System.Type.Empty Types را به عنوان پارامتر، بدان پاس می دهیم. در مواردی هم که به دنبال یک Constructor خاصی هستیم، بایستی آرایه ای از Type را تعیین نموده که هر کدام مشخص می کنند هر Constructor، چه پارامترهایی را دریافت خواهند کرد.

تا زمانی که یک رفرنس به تابع سازنده Constructor داریم، می توانیم به سادگی با فراخوانی متد ()invoke یک نمونه از کلاس TestClass را ایجاد نماییم. اگر نخواستیم پارامتری را به تابع ارسال کنیم، بایستی مقدار null را به متد ()invoke پاس دهیم. همچنین از متد ()GetMethod به همراه نام متدی که لازم داریم را برای گرفتن تابع ()TestMethod استفاده خواهیم کرد. در نهایت باز از ابزار کاربردی ()invoke برای فراخوانی تابع فوق، بهره می گیریم. اما این بار نیاز خواهیم داشت تا یک پارامتر از نوع آرایه از اشیا (array of objects) را تعیین کنیم.

اما به لطف Reflection، می توانید به صورت سریع مثلا عدد ۱۰ را به عنوان تنها پارامتر مورد نیاز ارسال کرده و سپس خروجی را پس از اجرا و فعال شدن متد مشاهده می کنیم.

درس ۴۹ : آموزش ایجاد یک کلاس Setting با Reflection در C#

آموزش ایجاد یک کلاس Setting Class با کمک Reflection در زبان C# :

بسیار خوب، در پایان این بخش قصد داریم تا با ارائه یک مثال کامل تر راجع به Reflection، آموزش ها را به پایان ببریم. مثال این درس، کمی بزرگتر از مثال های کل آموزش C# است، اما امیدواریم برای شما بسیار سودمند باشد. در خلال این مثال سعی شده تا مواردی که در بخش های قبل به آنها پرداختیم را نیز مرور کنیم.

یکی از سناریوهای رایج در هنگام تولید یک نرم افزار، امکان ذخیره و استفاده مجدد تنظیمات کاربر یا setting User است. هنگامی که چندین تنظیم یا setting دارید، طبیعی است که یک کلاس setting جهت خواندن و ذخیره انواع تنظیمات ایجاد کنید. همزمان که setting جدیدی را بخواهید به مجموعه خود اضافه کنید، بایستی متدهای ()Load و ()save را جهت ذخیره کردن این تنظیمات جدید، به روزرسانی کنید. اما چرا این امکان را فراهم نکنیم که خود کلاس setting خواص جدید را شناسایی کرده و آن ها را به صورت خودکار لود کند؟ به وسیله Reflection انجام این کار بسیار ساده خواهد بود و اگر شما مطالب سایر درس های این بخش را مطالعه کرده باشید، فهمیدن مثال این درس برای شما آسان است.

برای این که مثال درس را بتوانیم در حجم کمتری ارائه داده و فهمیدن آن ساده تر باشد، setting مربوط به یک شخص را به جای setting یک نرم افزار ذخیره خواهیم کرد که خلاصه تر است. اما شما می توانید مثال درس را به روش مورد نظر خود نیز تغییر دهید. خواهشمند است به این نکته دقت کنید که استفاده از Reflection، نسبت به خواندن و نوشتن Properties به صورت دستی، کندتر بوده و بایستی تصمیم بگیرید از کدام روش در هر برنامه استفاده کنید. این مسئله زمانی که سرعت اجرای نرم افزار برایتان مهم است، بسیار تاثیرگذار است. در مثال عملی این درس ما از یک فایل متنی text file برای مرتب سازی Sort چندین مقدار که با کاراکتر (/) از هم جدا شده اند، استفاده کرده ایم. اما در برنامه های واقعی، بهتر از ساختارهای مرتب دیتا مثل XML استفاده کنید. از طرف دیگر، باز در این مثال ما از سیستم مدیریت خطا در C# یا error handling استفاده نکرده ایم، در حالی که برای نرم افزارهای واقعی، مدیریت خطا حیاتی است.

خوب، بیایید مثال درس را شروع کنیم. در ابتدا، کلاس Person Class را داریم .که به راحتی می توانید آن را به کلاس setting تغییر دهید، تا کاربردی تر شود :

```
public class Person
    {
      private int age = -1;
      private string name = String.Empty;

      public void Load()
      {
        if (File.Exists("settings.dat"))
        {
          Type type = this.GetType();

          string propertyName, value;
          string[] temp;
          char[] splitChars = new char[] { '|' };
          PropertyInfo propertyInfo;

          string[] settings = File.ReadAllLines("settings.dat");
          foreach (string s in settings)
          {
            temp = s.Split(splitChars);
            if (temp.Length == 2)
            {
              propertyName = temp[0];
              value = temp[1];
              propertyInfo = type.GetProperty(propertyName);
              if (propertyInfo != null)
                this.SetProperty(propertyInfo, value);
            }
          }
        }
      }

      public void Save()
      {
```

```
    Type type = this.GetType();
    PropertyInfo[] properties = type.GetProperties();
    TextWriter tw = new StreamWriter("settings.dat");
    foreach (PropertyInfo propertyInfo in properties)
    {
      tw.WriteLine(propertyInfo.Name + "|" + propertyInfo.GetValue(this, null));
    }
    tw.Close();
  }

  public void SetProperty(PropertyInfo propertyInfo, object value)
  {
    switch (propertyInfo.PropertyType.Name)
    {
      case "Int32":
        propertyInfo.SetValue(this, Convert.ToInt32(value), null);
        break;
      case "String":
        propertyInfo.SetValue(this, value.ToString(), null);
        break;
    }
  }

  public int Age
  {
    get { return age; }
    set { age = value; }
  }

  public string Name
  {
    get { return name; }
    set { name = value; }
  }
}
```

البته حجم کد کلاس، کمی زیاد است، ولی به تشریح بخش های مختلف آن خواهیم پرداخت. در ابتدای کد کلاس، مجموعه ای از فیلدهای خصوصی Private fields داریم که برای نگهداری اطلاعات هر Person استفاده می شود. در انتهای کد کلاس هم، خواص عمومی (public Properties) قرار داده شده اند که برای خواندن و نوشتن فیلدهای خصوصی به کار می روند.

همچنین متد ()Load را داریم که به دنبال فایل setting.dot گشته و در صورت وجود داشتن، کل محتویات آن را خوانده و هر خط را در یک عضو از آرایه ای متنی قرار می دهد.

اکنون نگاهی به کدهای بخش setting بیاندازیم. این قسمت به دو بخش اصلی تقسیم می شود : اول یک خاصیت property name و بخش دوم مقدار یا value آن. اگر هر دو بخش وجود داشته باشد، از شی Type object برای گرفتن خاصیت به وسیله Property name استفاده کرده و سپس مقدار یا value را با استفاده از متد Set Property به آن پاس می دهیم.

متد ()SetProperty به type خاصیت یا Property که می خواهد تغییر کند، نگاه کرده و به صورت متقابل عکس العمل نشان می دهد. در حال حاضر، این متد فقط از نوع داده ای integer و string پشتیبانی می کند، اما شما می توانید پشتیبانی آن از سایر انواع داده ای را گسترش دهید.

متد ()save، آرایه ای از نسخه های مختلف PropertyInfo instance را دریافت کرده که هر کدام مربوط به یکی از خواص یا Properties تعیین شده برای Person است. سپس از یک TextWriter برای نوشتن هر خاصیت Property و مقدار آن در فایل data استفاده می کند.

اکنون فقط به کمی کد جهت استفاده ار کلاس نیاز داریم. مثال این درس، تلاش می کند تا اطلاعات person را از فایل setting خوانده و اگر موفق نشود، از کاربر می خواهد تا اطلاعات را وارد نماید :

```
class Program
    {
      static void Main(string[] args)
      {
        Person person = new Person();
        person.Load();
```

```
            if ((person.Age > 0) && (person.Name != String.Empty))
            {
              Console.WriteLine("Hi " + person.Name + " - you are " + person.Age + " years old!");
            }
            else
            {
              Console.WriteLine("I don't seem to know much about you. Please enter the following information:");
              Type type = typeof(Person);
              PropertyInfo[] properties = type.GetProperties();
              foreach (PropertyInfo propertyInfo in properties)
              {
                Console.WriteLine(propertyInfo.Name + ":");
                person.SetProperty(propertyInfo, Console.ReadLine());
              }
              person.Save();
              Console.WriteLine("Thank you! I have saved your information for next time.");
            }
            Console.ReadKey();
          }
        }
```

تقریبات تمامی کد مثال فوق، مطابق روند معمول است به جز بخشی که در آن از کاربر خواسته ایم تا اطلاعات را وارد کند. یک بار دیگر، از Reflection برای گرفتن خواص عمومی public properties از کلاس Person استفاده کرده ایم.

به شما پیشنهاد می کنیم، کلاس Person Class را برای دریافت اطلاعات بیشتر گسترش دهید. خواص Properties جدیدی را به فایل اضافه کرده و مشاهده کنید که این اطلاعات جدید در فایل ذخیره و لود می شود.

درس ۵۰ : جمع بندی آموزش C#

جمع بندی آموزش C# :

خسته نباشید. در طول ۵۰ درس آموزش گام به گام C#، سعی کردیم تا شما را با مفاهیم اصلی و پرکاربرد این زبان به روز و پیشرو آشنا کنیم. در پایان شما می توانید کلیه مطالب آموزشی درس را به صورت یک فایل کامل PDF به صورت رایگان از سوی آموزشگاه برنامه نویسی تحلیل داده، دانلود نموده و به مطالعه شخصی آن بپردازید.

همچنین در پایان هر درس، مثال های آن درس به صورت پروژه عملی، تهیه و اجرا شده و در ضمیمه درس ها قرار گرفته است.

با آموزش های بعدی در خدمت شما خواهیم بود.