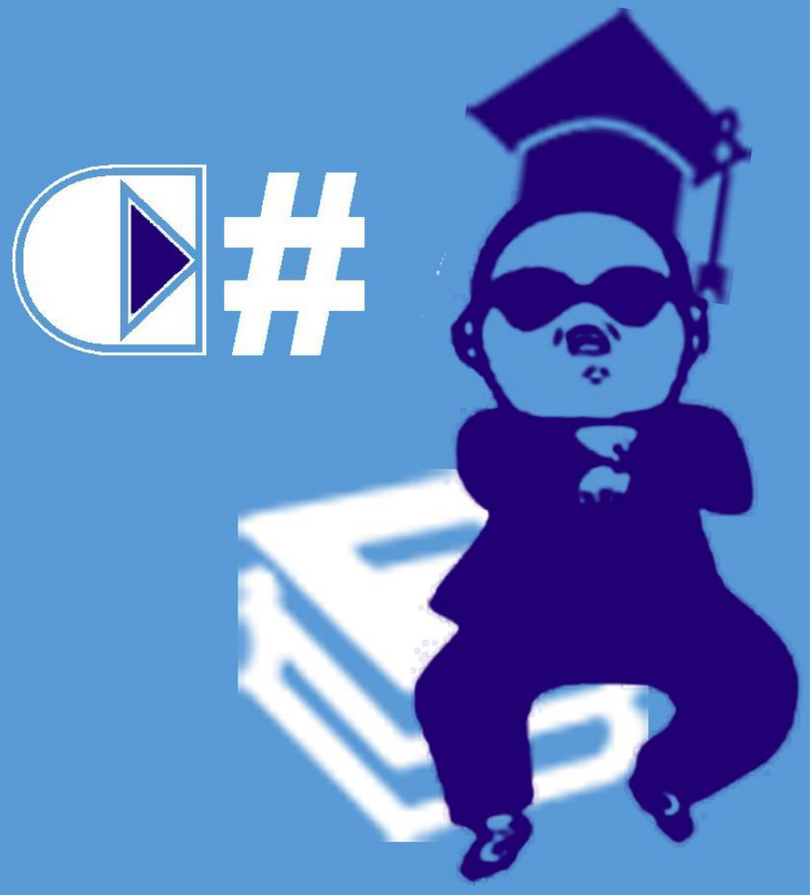
برنامه نویسی پیشرفته با سی‌شارپ



محمد فتحی
www.MiMFa.net

فهرست

Structs

ساختار استراکچر[1] بسیار شبیه کلاس[2] می‌باشد.، اما به طور کلی استراکچر ساختاری است که یک سری "داده" را تحت عنوان یک نام جدید گردآوری کرده و در حقیقت به نوعی یک تایپ جدید تعریف می‌کند که متشکل از ترکیب چند نوع داده دیگر است. مثلا یک استراکچر می‌تواند مجموعه ای از یک عدد صحیح یک عدد اعشاری و یک آرایه از کاراکتر ها باشد.

اما "کلاس" ساختاری کاملا متفاوت با استراکچر دارد. کمترین تفاوت یک استراکچر و یک کلاس در این است که استراکچر معمولا بدون تابع می‌باشند؛ در حالی که توابع، اجزا جدایی ناپذیر کلاس ها هستند.

---

[1] struct

[2] class

بنابراین به بیانی ساده تر؛ **از استراکچر نمی‌توان انتظار انجام عملی را داشت**، چرا که تنها ظروفی برای نگهداری داده ها هستند. در حالی که یک شئ[1](که توسط "کلاس" پیاده سازی می‌شود) یک موجود کاملا مستقل است که حتی می‌تواند طوری برنامه ریزی شود که صرف ایجاد شدن، وظیفه خاصی را به انجام برساند؛ بدین معنا که حتی اگر فقط یک نمونه[2]از کلاس ساخته شود، سازنده کلاس می‌تواند یک سری عملیات و محسابیات را انجام دهد. (درون کانستراکتور یا سازنده کلاس می‌توان مشخص کرد که یک سری کد ها در زمان ایجاد یک نمونه از کلاس اجرا شوند و عملیاتی را انجام دهند)

---

[1] object

[2] instanse

به طور کلی ویژگی های استراکچر را می‌توان به صورت زیر برشمرد:

- با لغت کلیدی «struct» تعریف می‌شود

```
public struct Strct {...}
```

- می‌تواند بدون کلمه کلیدی new ساخته شود

```
Strct test = [value];
```

- نوع داده‌ای[1]است و شیء بوجود آمده از آن در استَک[2]ذخیره می‌شود .
- نمی‌تواند ارث بری کند
- زمانی که از آرایه ها استفاده می‌شود مناسب تر است.
- نمی‌توان به متغیرهای نمونه و عضو مقدار اولیه داد .

---

[1] Value type: مانند int ، float و... که برای استفاده از آنها، نیازی نیست با کمک کلمه کلیدی new یک نمونه ساخته شود.

[2] Stack

- نمی‌تواند ایستا[1] باشد.
- نمی‌تواند دارای هرگونه سازنده‌ی[2] بدون پارامتر، یا حتی سازنده پیشفرض-که CLR فراهم کرده باشد.
- از مخرب ها[3] پشتیبانی نمی‌کند.
- نمی‌تواند مقدار null داشته باشد.

نکته: به طور کلی می‌توان گفت استراکچر یک قالب داده ای می‌باشد که می‌تواند بسته به نوع داده ها آن را شخصی سازی نمود.

---

[1] static

[2] constructor

[3] destructor

Access Modifiers

اصولا کاربرد سطوح دسترسی[۱] بر روی دو حوزه می‌باشد.

- در تعریف کلاس،[۲] اینام[۳] یا ساختارها.[۴]
- در تعریف متغیرها،[۵] توابع،[۶] سازنده‌ها[۷] و ...

---

۱ Access Modifiers

۲ Class

۳ Enum

۴ Structure

۵ Variable

۶ Method

۷ Constructor

روش های دسترسی به متدها و خاصیتهای تعریف شده در داخل یک کلاس به پنج دسته تقسیم می‌شود:

۱. public

۲. private

۳. protected

۴. internal

۵. protected internal یا internal protected

۱. public

متد ها و خاصیت هایی که در یک کلاس به صورت عمومی[۱] تعریف شوند، هر کلاسی که از این کلاس یک نمونه می سازد یا از این کلاس ارث بری دارد ، می‌تواند به این عناصر دسترسی داشته باشد.

بطور کلی این سطح دسترسی هیچ گونه محدودیتی قائل نیست. امکان استفاده از آن برای کلاس‌ها (آیتم‌های اول) و متغییرها و ... (آیتم‌های دوم) وجود دارد؛ و زمانی که یک کلاس عمومی‌تعریف می‌شود بدین معناست که هر کس (چه از داخل پروژه و چه خارج از پروژه) امکان استفاده از آن کلاس را دارد, و وقتی متغییر یا متدی با این سطح دسترسی تعریف می‌شود هر کسی که به کلاس دسترسی دارد می‌تواند از آن متغییر یا متد استفاده نماید.

نکته: سطح دسترسی عمومی برای گروه آیتم‌های اول (کلاس‌ها، استراکچرها و...) نیز قابل استفاده می باشد.

---

[۱] public

۲. private

متغییر و یا متد و ... که به صورت خصوصی[۱]تعریف شود, فقط و فقط داخل همان کلاس قابل استفاده خواهدبود.

نکته: سطح دسترسی خصوصی برای گروه آیتم‌های اول (کلاس‌ها، استراکچرها و...) قابل استفاده نیست.

---

[۱] private

۳. protected

متدها و خاصیت‌هایی که در یک کلاس به صورت محافظت شده[۱] تعریف شوند، در داخل همان کلاس و کلاس‌هایی که از این کلاس مشتق شده اند (ارث برده اند) قابل دسترس هستند و نمونه های ایجاد شده از کلاس به این متدها و خاصیت‌ها دسترسی ندارند.

---

[۱] protected

به عنوان مثال کلاس Person دارای یک متد به نام GetInfo است.

```
public class Person
{
   public int Age;
   public string Name;
   protected void GetInfo()
   {
      Console.WriteLine("Name:");
      this.Name = Console.ReadLine();
      Console.WriteLine("Age:");
      this.Age = int.Parse(Console.ReadLine());
   }
}
```

حال کلاس Emp که از کلاس Person به ارث رفته است می‌خواهد از متد GetInfo استفاده کند, چون متد GetInfo به صورت protected تعریف شده -در نتیجه تمامی‌کلاس‌هایی که از Person به ارث بروند امکان استفاده از آن را دارند- قادر به انجام این کار است. در صورتیکه از هیچ کجای دیگر امکان استفاده از این متد برای کلاس‌های دیگر مثل کلاس Program وجود ندارد!

نکته: متد محافظت شده از بیرون دیده نمی‌شود.

نکته: سطح دسترسی مخافظت شده برای گروه آیتم‌های اول (کلاس‌ها، استراکچرها و...) قابل استفاده نیست.

۴. internal

کلاسها یی که به صورت داخلی[۱] تعریف شوند، تنها کلاسهایی می‌توانند از آنها استفاده کنند که در یک اسمبلی[۲] یکسان تعریف شده باشند.

به عنوان مثال بسیاری از مواقع پیش می‌آید کلاسی را ایجاد می نمایید که احتمال دارد در پروژه‌های دیگری بیرون از پروژه جاری استفاده شود. (مثلا در مورد برنامه نویسی چندلایه). حالا فرض نمایید که نمی‌خواهید یک کلاس یا متغییر یا ... آن کلاس در اختیار کسانی قرار بگیرد که بیرون از پروژه جاری از این اسمبلی استفاده می‌کنند. (مثلا یک Component را در نظر بگیرید که قرار است داخل n پروژه دیگر استفاده شود).

برای همین می‌توانید با استفاده از internal فقط به کلاس‌هایی که داخل این پروژه شما هستند اجازه دهید که از این کلاس یا متغییر یا ... استفاده کنند.

---

[۱] internal

[۲] Assembly

مثال:

```
// Assembly1.cs
   internal class BaseClass
   {
      public static int IntM = 0;
   }
// Assembly2.cs
   class TestAccess
   {
      public static void Main()
      {
         BaseClass myBase = new BaseClass(); // با خطا مواجه می‌شود
      }
   }
```

در مثال قبل کلاس BaseClass در اسمبلی ۱ به صورت internal تعریف شده است ، کلاس TestAccess تعریف شده در اسمبلی ۲ یک نمونه از کلاس BaseClass را ایجاد می‌کند که این یک خطا محسوب می‌شود.

نکته: گروه آیتم‌های اول (کلاس‌ها، استراکچرها و...) نیز به صورت پیشفرض داخلی می باشند.

نکته: سطح دسترسی داخلی برای گروه آیتم‌های اول (کلاس‌ها، استراکچرها و...) نیز قابل استفاده می باشد.

۵. internal protected

این سطح تلفیقی است از internal و protected؛ به این معنا که اگر متغییری به صورت داخلی محافظت شده[۱]تعریف شده باشد. کلاس هایی که داخل این پروژه هستند و یا از کلاسی که این متغییر درونش قرار دارد به ارث رفته باشند, اجازه دارند که از این متغییر استفاده نمایند.

نکته: سطح دسترسی داخلی محافظت شده برای گروه آیتم‌های اول (کلاس‌ها و...) قابل استفاده نیست.

---

[۱] internal protected

Friend Assemblies

همانطور که می‌دانید در زبان C++ ، در مبحث شئ گرایی[۱] مفهومی‌بنام دوستی[۲]وجود داشت، بدین معنا که یک کلاس دوست کلاس دیگر می‌شد و به اجزای اطلاعاتی[۳]آن کلاس دسترسی عمومی[۴]پیدا می‌کرد ، مشابه این گونه امکان در سی شارپ نیز وجود دارد، و همانگونه که در مثال زیر مشاهده خواهید کرد، نحوه تعریف آن با C++ اندکی متفاوت است.

---

۱ OOP

۲ Friend

۳ Data member

۴ Public

```
// AssemblyA
using System.Runtime.CompilerServices;
using System;
[assembly: InternalsVisibleTo("AssemblyB")]//ایجاد رابطه دوستی
// کلاس که به صورت پیشفرض داخلی می‌باشد
class FriendClass
{
   public void Test()
   {
      Console.WriteLine("Sample Class");
   }
}
```

```
// کلاس دیگر که به صورت عمومی‌تعریف شده
public class ClassWithFriendMethod
{
   internal void Test()
   {
      Console.WriteLine("Sample Method");
   }
}
```

لازم است بدانید اکنون در پروژه ای دیگر که نام اسمبلی آن AssemblyB می‌باشد نیز براحتی می‌توان به هر دو کلاس و هر دو متد های تعریف شده آن ها دسترسی عمومی پیدا کرد.

نکته: از این ترفند می‌توان برای افزایش امنیت در امور لایه بندی استفاده نمود.

Accessibility Capping

به کلاس زیر توجه فرمایید...

```
internal class C
{
    public void Foo()
    {
    //....
    }
}
```

در اینجا یک نمونه کلاس موجود است که با سطح‌دسترسی داخلی ایجاد شده درحالی‌که اجزای داخل آن -که در اینجا یک متد است- با سطح دسترسی بالاتر یعنی عمومی‌تعریف شده اند.

به این اتفاق اصطلاحا سرپوش دسترسی[1] گفته می‌شود؛ بدین معنا که اکنون اجزای این کلاس چه از نوع عمومی و چه از نوع داخلی تعریف شوند فقط و فقط سطح دسترسی به آنها به صورت داخلی می‌باشد زیرا که سطح دسترسی آیتم بالایی آن ها داخلی می‌باشد.

مزیت این عمل زمانی مشخص می‌شود که بنا به هر دلیلی قصد تغییر دسترسی اجزای داخلی و کلاس را داشتیم؛ در این حالت فقط کافی است سطح دسترسی کلاس را به دلخوه تغییر دهیم، تا به سایر اجزا هم همان دسترسی را داشته باشیم.

[1] Accessibility Capping

Interfaces

میانجی یا واسط[۱] به عنوان یک قرارداد نحوی تعریف می‌شود که همه ی گروه هایی پیرو باید آن را دنبال کنند. واسط قسمت «چه؟[۲]» از یک قرارداد نحوی را تعریف می‌کند و گروه های مشتق قسمت «چگونه؟[۳]» از این قراردادها را تعریف می‌کنند.

به بیانی صریحتر؛ واسط ها؛ ویژگی‌ها[۴]، متدها[۵] و رویدادهایی[۶] را تعریف می‌کنند که اعضای آن می‌باشند.

---

۱ Interfaces

۲ What?

۳ How?

۴ Property

۵ Method

۶ Event

واسط تنها شامل اعلام اعضا می‌باشد. تعریف اعضا مسئولیت گروه مشتق شده می‌باشد؛ که اغلب به تامین یک ساختار استاندارد کمک می‌کند که گروه های مشتق دنبال می‌کنند.

گروه های نظری نیز تا حدی همین هدف را دارند، به هرحال عمدتا زمانی استفاده می‌شوند که تنها تعدادی از متدها قرار است به وسیله ی گروه پایه اعلام شوند و گروه مشتق شده قابلیت ها را اجرا می‌کند.

واسط ها با استفاده از کلمه ی کلیدی interface اعلام می‌شوند، که شبیه اعلام یک گروه می‌باشد. بیانیه های واسط به طور پیش فرض عمومی می‌باشند.

در زیر مثالی از اقلام واسط ها را می‌بینید.

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
}
internal class Countdown : IEnumerator
{
    int count = 11;
    public bool MoveNext() => count-- > 0 ;
    public object Current => count;
}
```

و اما در واسط ها نکاتی دیگر نیز وجود دارد که در زیر به برخی از آنها می پردازیم:

- اجزای داخل آن بدون مشخص نمودن سطح دسترسی تعریف می‌شوند.
- نمی‌توان در آن فیلد تعریف نمود.
- کلاسی که از واسط ارث‌بری می‌کند باید تمامی متدهای آن را دقیقا همان گونه که در واسط مشخص شده پیاده‌سازی نماید. به بیان کلی، فرم و ساختار کلی خود را از واسط می‌گیرد و نحوه رفتار و پیاده‌سازی آنرا خود انجام می‌دهد.
- می‌توان از ارث بری برای واسط ها با واسط های والد خود استفاده نمود.

```
public interface IUndoable { void Undo(); }

public interface IRedoable : IUndoable { void Redo(); }
```

- هر ارث بر می‌تواند از چند واسط استفاده نماید.

- هر جزء می‌تواند در واسطهای متفاوت با ساختارهای متفاوت ظاهر شده؛ و به هنگام صدا زدن مجزا شود.

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }
public class Widget : I1, I2
{
   public void Foo() // Implicit implementation
   {
      Console.Write("Widget's implementation of I1.Foo");
   }
   int I2.Foo() // Explicit implementation of I2.Foo
   {
      Console.Write("Widget's implementation of I2.Foo");
      return 42;
   }
}
```

- نمی‌توان در آن هیچ چیز را مقدار دهی نمود.
- واسط ها در حقیقت نوع مرجع[۱] می‌باشند.
- نمی‌توان یک واسط را بصورت virtual اعلان نمود.
- هدف از ایجاد یک واسط تعیین توانائی‌هایی است که می‌خواهیم در یک کلاس وجود داشته باشند.
- از روی یک واسط نمی‌توان نمونه‌ای جدید ایجاد کرد بلکه باید کلاسی از آن ارث‌بری نماید.
- ارث‌بری از کلاس رابطه "است" یا "بودن"[۲] را ایجاد می‌کند (ماشین یک وسیله نقلیه است) ولی ارث‌بری از یک واسط نوع خاصی از رابطه، تحت عنوان "پیاده‌سازی"[۳] را ایجاد می‌کند. ("می‌توان ماشین را با وام بلند مدت خرید" که در این جمله ماشین می‌تواند خریداری شدن بوسیله وام را پیاده‌سازی کند.)

---

[۱] Reference Type

[۲] is-a relation

[۳] implement relation

فرم کلی اعلان interface ها بشکل زیر است :

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

attributes : صفتهای واسط

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

interface-name : نام واسط

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

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

Delegates

می‌توان گفت که نماینده[1] یکی از ویژگی های جذاب زبان برنامه نویسی سی شارپ بوده که امکانات بالقوه فوق‌العاده‌ای را به این زبان می‌آفزاید.

نماینده ها نوع‌هایی[2] هستند که اشیاء آن ها می توانند متدهای کلاس‌های دیگر و متدهای اشیاء دیگر را فرخوانی کنند. در واقع یک شیء از یک نماینده برای فراخوانی متدهای کلاس‌ها و اشیاء دیگر ایجاد می‌شود.

پس در تعریفی روان تر؛ نماینده کلاسی است که اشیاء ساخته شده از آن می‌توانند توابع ثبت شده[3] در خود را به ترتیب اجرا نمایند!

برای استفاده از نماینده چهار مرحله اصلی وجود دارد که در ذیل، هر مرحله با مثال مشخص خواهد شد:

---

[1] Delegate

[2] type

[3] Register

مرحله اول: تعریف نماینده[1]

در این مرحله اقدام به تعریف کلاس نماینده با کلمه کلیدی delegate می‌کنیم:

```
public delegate void MyDelegate(int n);
```

معنی عبارت فوق این است که ما می‌خواهیم یک کلاس نماینده تعریف کنیم که اشیاء آن بتوانند توابعی را در داخل خود ثبت کنند که پارامتر ورودی آنها یک عدد صحیح (int n) بوده و پارامتر خروجی نداشته باشند.[2]

---

[1] Delegate Definition

[2] void

برای روشن شدن مطلب کلاسی به نام Employee و به شکل ذیل تعریف می نماییم:

```
public class Employee
{
    public int Age;
    public string FullName;

    public Employee(string fullName, int age)
    {
        Age = age;
        FullName = fullName;
    }

    public void DoIt(int n)
    {
        System.Console.WriteLine("هستم من " + FullName + " می‌کنم کار و " + n + " روز در ساعت .");
    }
}
```

همانگونه که مشاهده می‌کنید، ما به طور عمدی در این کلاس تابعی تعریف کرده ایم (DoIt) که پارامترهای ورودی و خروجی آن با آنچه که در تعریف delegate عنوان گردیده است، مطابقت داشته باشد.

مرحله دوم: ساخت یک نمونه[1]

در این مرحله نسبت به ایجاد یک شیء از کلاس MyDelegate اقدام می‌کنیم:

```
MyDelegate DelegateInstance;
```

در این دستور، ما یک شیء به نام DelegateInstance از کلاس MyDelegate تعریف نموده ایم.

حال برای ادامه مسیر، از کلاس Employee یک شیء به نام oEmployee به شکل ذیل ایجاد می‌کنیم:

```
Employee oEmployee = new Employee("علی علوی", ۲۷);
```

[1] Create Instance

مرحله سوم: مقداردهی

در این مرحله تنها کافی است که تابع DoIt شیء oEmployee را در شیء DelegateInstance به شکلی که در ذیل ذکر گردیده است ثبت نماییم:

```
DelegateInstance = new MyDelegate(oEmployee.DoIt);
```

مرحله چهارم: صدازدن[1]

در این مرحله با صدا زدن شیء DelegateInstance همراه با یک پارامتر عددی، تابع ثبت شده در داخل آن به همان پارامتر عددی مشخص شده اجرا می‌شود. توجه فرمایید که در این مثال تنها یک تابع ثبت شده در داخل شیء Delegate وجود دارد.

```
DelegateInstance(5);
```

کاملا واضح است که به راحتی می‌توانستیم پس از ایجاد شی، oEmployee، با اجرا کردن تابع Doit همراه با همان پارامتر عددی، به همان نتیجه نائل آییم. اما دقت فرمایید که همیشه این چهار مرحله به این شکلی که در اینجا مطرح گردیده است در کنار هم قرار نمی‌گیرند.

نکته: در یک پروژه واقعی، این چهار مرحله، هر کدام در یک قسمت از برنامه تعریف و بکارگرفته می‌شوند و این مساله امکانات بسیار مفید و جذابی را برای زبان برنامه نویسی سی شارپ به ارمغان می‌آورد.

---

[1] Call

- با تعریف نماینده برای توابع لایه های دیگر می‌توان به مراتب از میزان وابستگی لایه ها به یکدیگر کاست.
- تعریف بیش از یک متد (با ساختار شبیه به هم) داخل یک نماینده. برای این کار به جای استفاده از = موقع new کردن از =+ استفاده خواهیم کرد.
- نماینده ها نمی‌توانند ایستا تعریف شوند.

- اجرای یک متد در نخ[۱]سازنده فرم (برای جلوگیری از تداخل در برنامه نویسی چند نخی[۲])

```
delegate void SampleDelegate();
private void SampleMthod() {}
public Class1(Form form) // سازنده کلاس Class۱
{
  if (form.InvokeRequired)
  {
    SampleDelegate d = new SampleDelegate (SampleMthod);
    form.Invoke(d);
  }
}
```

---

۱ Thread

۲ Multi-Thread-Programming

- اجرای غیرهمگام[1] متدها

```
delegate void SampleDelegate();
private void SampleMethod() { }
public void Main()
{
  SampleDelegate d = new SampleDelegate(SampleMethod);
  d.BeginInvoke(SampleMethodCallback, null);
}
private void SampleMethodCallback(IAsyncResult result) { }
```

[1] Asynchronous

- تشکیل مجموعه ای از متدهای با امضای یکسان بدون محدودیت در تعلق داشتن به کلاس خاص

```
delegate void HandlerDelegate();
class Dispatcher
{
   private Dictionary<string, HandlerDelegate> _handlers;
   public void AddHandler(string eventt, HandlerDelegate d)
   {
      _handlers.Add(d);
   }
   public Dispatcher()
   {
      _handlers = new Dictionary<string, HandlerDelegate>();
   }
   public void Dispatch(string eventt)
   {
      if (_handlers.ContainsKey(eventt))
         _handlers[eventt].Invoke();
   }
}
```

- برای ایجاد متدهای بی نام[۱] که در .NET 3.5 و LINQ کاربرد فراوانی به شکل «عبارت لامبدا[۲]» دارد.
- طبق تکنیک هم پراکنشی[۳] متدهایی که امضای یکسانی با امضای تعریف شده در نماینده دارند می‌توانند نوع بازگشتی‌شان متفاوت از نماینده باشد اما این نوع بازگشتی باید زیر نوعی از نوع تعریف شده در نماینده باشد.

```
public class Mammal{}
public class Dog : Mammal{}
delegate Mammal SampleDelegate();

public Dog SampleDogMethod(){}
```

---

[۱] Anonymous Method

[۲] Lambda Expression

[۳] Covariance

- متدهایی که نوع پارامترهای ورودیشان والد نوع پارامترهای ورودی مشخص شده در نماینده باشد، نیز در نماینده پذیرفته می‌شوند.

```
public class Mammal { }
public class Dog : Mammal { }
public delegate void SampleDelegate(Dog dog);

public void SampleMammalMethod(Mammal mammal){}
```

- در شرایطی که داده هائی موجود است اما در زمان برنامه نویسی یا به عبارت صحیح تر در زمان کامپایل دقیقا مشخص نیست که چه عملیاتی باید روی آن ها انجام شود.

نکته: می‌توان نماینده را از نوع کلی[1] تعریف نمود.

```
public delegate T Transformer<T> (T arg);
static double Square (double x) => x * x;
static void Main()
{
      Transformer<double> s = Square;
      Console.WriteLine (s (3.3)); // 10.89
}
```

[1] Generic

Events

رابط گرافیکی کاربر در دات نت و ویژوال سی شارپ از مکانیزم کنترل کننده رویداد[1]برای کنترل رویدادها که در هنگام اجرای برنامه به وقوع می پیوندند استفاده می‌کند.

.رویدادها رفتارهایی یا اتفاقاتی هستند که هنگام اجرای برنامه به وقوع می پیوندند.کنترل رویداد فرایند نظارت بر وقوع یک رویداد مشخص می‌باشد.

فرم های ویندوزی از کنترل کننده رویداد برای اضافه کردن یک قابلیت و پاسخ به کاربر استفاده می‌کنند. بدون کنترل کننده رویداد، فرم ها و رابط گرافیکی تا حد زیادی بدون استفاده هستند. رویدادها با استفاده از یک نماینده[2] به عنوان یک نوع تعریف می‌شوند. نماینده ها آدرس متدها را در خود ذخیره می‌کنند.

---

[1] Event

[2] delegate

در مثال زیر یک نماینده و یک رویداد تعریف شده اند.

```
public delegate void SampleEventHandler(int);

public event SampleEventHandler SampleEvent;
```

بر اساس تعریف بالا نماینده آدرس متدهایی را قبول می‌کند که دارای مقدار برگشتی نیستند و یک آرگومان ساده از نوع عدد[1]قبول می‌کنند .سپس از این نوع[2]نماینده برای ایجاد رویداد مورد نظر استفاده می‌کنیم. حال یک کنترل کننده رویداد به رویداد اضافه می‌کنیم. (کنترل کننده های رویداد متدهای متناظر با نوع نماینده رویداد هستند) و هنگام وقوع رویداد اجرا می‌شوند. آنها (کنترل کننده های رویداد) چسبیده به یک رویداد هستند و هنگامی‌که رویداد به وقع می پیوندد اجرا می‌شوند.

---

[1] int

[2] Type

می‌توان چندین کنترل کننده‌ی رویداد را با += به یک رویداد متصل کرد تا هنگام وقوع رویداد اجرا شوند. برای این کار ابتدا باید کنترل کننده رویداد را ایجاد کنید.

بعد از ایجاد ،مطمئن شوید که امضای آن با نماینده‌ای که رویداد از آن استفاده می‌کند مطابق باشد. به عنوان مثال به نماینده‌ی مثال بالا توجه کنید که نوع برگشتی نداشته[1]و یک پارامتر از نوع عدد ورودی می‌گیرد، پس کنترل کننده رویداد ما نیز باید دارای نوع برگشتی خالی و یک پارامتر از نوع عدد باشد (به این نکته توجه کنید که سطح دسترسی مهم نیست).

```
public void ShowMessage(int number)
{
    MessageBox.Show("Hello World");
}
```

سپس می‌توان با استفاده از عملگر += یک رویداد را متصل کرد :

```
SampleEvent += new SampleEventHandler(ShowMessage);
```

---

[1] void

برای فعال کردن رویداد به همان روشی که متد ها را فراخوانی می‌کردیم ان را صدا زده و آرگومانی را که لازم دارد به آن ارسال می‌کنیم.

```
ShowMessage(۳);
```

در زیر مثالی دیگر و مجتمع از رویداد را مشاهده می فرمایید:

```
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);
public class Stock
{
    string symbol; decimal price;
    public Stock(string symbol) { this.symbol = symbol; }
    public event PriceChangedHandler PriceChanged;
    public decimal Price
    {
        get { return price; }
        set
        {
            if (price == value) return;
            if (PriceChanged != null)
                PriceChanged(price, value);
            price = value;
        }
    }
}
```

الگوی استاندارد رویدادها

به طور کلی هر رویداد بهتر است که دو مولفه‌ورودی داشته باشد؛ یکی بیانگر خود مولفه ای که باعث ایجاد این رویداد شده و دیگری مولفه ای که مقادیر جدید مولفه را نمایش می‌دهد.

جهت توضیح این مبحث یک مثال طرح نموده و طی آن توضیحات لازم داده می‌شود.

```
//کلاس ارائه دهنده ی ارگومان های تغیرات قیمت
public class PriceChangedEventArgs : EventArgs
{
    public readonly decimal LastPrice, NewPrice; //مولفه ها
    public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
//سازنده
    {
        LastPrice = lastPrice; NewPrice = newPrice;
    }
}
```

```
public class Stock //کلاس اصلی با نام موجودی
{
    string symbol; decimal price;
    public Stock(string symbol) { this.symbol = symbol; }
    //رویداد تغییر قیمت
    public event EventHandler<PriceChangedEventArgs> PriceChanged;
    //تابع اجرای رویداد
    protected virtual void OnPriceChanged (PriceChangedEventArgs e)
    {
        if (PriceChanged != null) PriceChanged(this, e);
    }
    public decimal Price
    {
        get { return price; }
        set
        {
            if (price == value) return;
             //صدا زدن تابع در صورت تنظیم ثیمت
            OnPriceChanged(new PriceChangedEventArgs(price, value));
            price = value;
        }
    }
}
```

و جهت استفاده از آن می‌توان به صورت زیر عمل نمود:

```
static void Main()
{
    Stock stock = new Stock ("ریال");
    stock.Price = 27.10M;
    stock.PriceChanged += stock_PriceChanged;
    stock.Price = 31.59M;
}
static void stock_PriceChanged (object sender, PriceChangedEventArgs e)
{
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
    Console.WriteLine ("توجف، ۱۰ درصد قیمت افزایش یافت");
}
```

نکته: به مثال زیر توجه فرمایید:

```
EventHandler priceChanged; // Private delegate
public event EventHandler PriceChanged
{
    add { priceChanged += value; }
    remove { priceChanged -= value; }
}
public void Test()
{
    PriceChanged += new EventHandler((object o, EventArgs e) => { });
    PriceChanged -= new EventHandler((object o, EventArgs e) => { });
}
```

همانگونه که مشاهده می فرمایید مطابق مثال فوق می‌توان امکان افزودن و یا کاستن از رویداد های مورد نظر را شخصی سازی نمود.

معرفی رویدادهای فرم در سی شارپ

وقتی نمونه ای از برنامه ی ویژوال ایجاد می‌کنیم، اولین مرحله در برنامه نویسی، ایجاد فرم و اضافه کردن کنترلها به آن است. فرم ، بستری برای دربرگرفتن کنترلها جهت برنامه نویسی ویژوال است و کنترل، قطعات نرم افزاری هستند که قابلیت استفاده مجدد را دارند. فرم‌های برنامه با استفاده از کنترل ها طراحی می‌شوند. هر فرم دارای تعدادی رویداد است که در شرایط خاصی رخ می‌دهند. بعضی از رویدادهای مهم فرم عبارتنداز:

رویداد Activated : وقتی رخ می‌دهد که فرم فعال شود.

رویداد BackgroundColorChanged : وقتی رخ می‌دهد که رنگ زمینه فرم عوض شود.

رویداد BackgroundImagedChanged : وقتی رخ می‌دهد که تصویر مربوط به زمینه فرم عوض شود.

رویداد BindingContextChanged : وقتی رخ می‌دهد که خاصیت BindingContext تغییر کند.

رویداد CausesValidationChanged : وقتی رخ می‌دهد که خاصیت CausesValidation تغییر کند.

رویداد ChangeUICaues: وقتی رخ می‌دهد که صفحه کلید عوض شود.

رویداد Click: وقتی رخ می‌دهد که کنترل کلیک شود.

رویداد FormClosed: وقتی رخ می‌دهد که فرم بسته شود.

رویداد FormClosing: وقتی رخ می‌دهد که فرم در حالت بسته شدن باشد.

رویداد ContextMenueStripChanged: وقتی بر روی کنترلی کلیک راست کنید، منویی ظاهر میشود که منوی میانبر نام دارد.

رویداد CursorChanged : وقتی رخ می‌دهد که شکل مکان نما تغییر کند.

رویداد Deactive : هنگامی‌که کنترل غیر فعال میشود، این رویداد رخ می‌دهد.

رویداد DockChanged : وقتی رخ می‌دهد که الحاق تغییر کند.

رویداد DoubleClick: وقتی رخ می‌دهد که کنترل کلیک مضاعف شود.

رویداد DragDrop: وقتی رخ می‌دهد که کنترلی بر روی این کنترل کشیده شود و رها گردد.

رویداد DragEnter: وقتی رخ می‌دهد که کنترلی از طریق کشیده شدن وارد فرم شود.

رویداد DragLeave: وقتی رخ می‌دهد که کنترلی از طریق کشیدن از فرم خارج شود.

رویداد DragOver: وقتی رخ می‌دهد که کنترلی بر روی فرم کشیده شود.

رویداد EnablesChanged : وقتی که خاصیت Enabled مربوط به کنترلی تغییر یابد، این رویداد رخ می‌دهد.

رویداد Enter : وقتی رخ می‌دهد که مکان نما به فرم وارد شود.

رویداد FontChanged : وقتی رخ می‌دهد که خاصیت Font کنترلی تغییر یابد.

رویداد ForeColorChanged : وقتی رخ می‌دهد که رنگ متن تغییر یابد.

رویداد GiveFeedback : در مدت زمان کشیدن کنترل این رویداد رخ می‌دهد.

رویداد HelpRequested : وقتی رخ می‌دهد که کاربر برای کنترلی درخواست کمک کند.

رویداد ImeModeChanged : وقتی رخ می‌دهد که حالت ویراستار متد ورودی تغییر یابد.

رویداد InputLanguageChanged : وقتی رخ می‌دهد که زبان صفحه کلید تغییر کند.

رویداد KeyDown : وقتی رخ می‌دهد که کلیدی از صفحه کلید فشرده شود.

رویداد KeyPress : وقتی که کلیدی فشرده شود ، این رویداد رخ می‌دهد. رویداد KeyPress قبل از رویداد KeyDown رخ می‌دهد.

رویداد KeyUp : وقتی که کلید فشرده شده رها شود ، این رویداد رخ می‌دهد.این رویداد بعد از رویدادهای KeyPress و KeyDown رخ می‌دهد.

رویداد Layout : وقتی رخ می‌دهد که کنترلی موقعیت کنترل های روی خودش را تغییر می‌دهد.

رویداد Leave : وقتی رخ می‌دهد که مکان نما کنترلی را ترک کند.

رویداد Load : وقتی کنترلی بار میشود، این رویداد رخ می‌دهد.

رویداد LocationChanged : وقتی رخ می‌دهد که مکان کنترلی تغییر کند.

رویداد MaximizidBoundsChanged : وقتی رخ می‌دهد که مقدار خاصیت MaximizidBounds تغییر کند.

رویداد MdiChildActivate : اگر چند سند در برنامه داشته باشیم وقتی که یک فرم فرزند MDI غیرفعال شده یا بسته میشود، این رویداد رخ می‌دهد.

رویداد MenuComplete : وقتی رخ می‌دهد که نمایش منو کامل شود.

رویداد MenuStart : وقتی رخ می‌دهد که نمایش منو شروع شود.

رویداد MinimimSizeChanged : وقتی رخ می‌دهد که خاصیت MinimimSize تغییر کند.

رویداد MouseDown : وقتی رخ می‌دهد که کلید ماوس کلیک شود.

رویداد MouseEnter : وقتی رخ می‌دهد که مکان نما وارد کنترلی شود.

رویداد MouseHover : وقتی رخ می‌دهد که مکان نما منتظر باشد.

رویداد MouseLeave : وقتی رخ می‌دهد که مکان نما کنترلی را ترک کند.

رویداد MouseMove : وقتی رخ می‌دهد که مکان نما بر روی کنترلی حرکت کند.

رویداد MouseUp : وقتی رخ می‌دهد که دکمه ماوس رها شود.

رویداد Move : وقتی رخ می‌دهد که کنترلی شروع به حرکت کند.

رویداد Paint : وقتی رخ می‌دهد که کنترلی رسم شود.

رویداد ParentChanged : وقتی رخ می‌دهد که خاصیت Parent کنترل تغییر یابد.

رویداد QueryContinueDrag : این رویداد در زمان کشیده شدن و رها کردن کنترل اتفاق می‌آفتدو مشخص میکند آیا کنترل منبع کشیده شده میتواند کشیدن و رها کردن را لغو کند یا خیر.

رویداد Resize : وقتی رخ می‌دهد که اندازه کنترل تغییر میابد.

رویداد RightToLeft : وقتی رخ می‌دهد که مقدار خاصیت RightToLeft تغییر کند.

رویداد SizeChanged : وقتی رخ می‌دهد که مقدار خاصیت Sizeتغییر کند.

رویداد StyleChanged : وقتی رخ می‌دهد که سبک کنترل تغییر کند.

رویداد SystemColorsChanged : وقتی رخ می‌دهد که رنگ های سیستم تغییر کند.

رویداد TabIndexChanged : وقتی رخ می‌دهد که مقدار خاصیت TabIndexتغییر کند.

رویداد TabStopChanged : وقتی رخ می‌دهد که مقدار خاصیت TabStopتغییر کند.

رویداد TextChanged : وقتی رخ می‌دهد که متن کنترل تغییر کند.

رویداد Validated: پس از اعتبارسنجی داده ها رخ می‌دهد.

رویداد Validating : در هنگام اعتبارسنجی داده ها رخ می‌دهد.

رویداد VisibleChanged : وقتی رخ می‌دهد که خاصیت Visible کنترل تغییر کند.

رویداد AutoSizeChanged : وقتی رخ می‌دهد که خاصیت AutoSize تغییر کند.

رویداد AutoValidatedChanged : وقتی رخ می‌دهد که خاصیت AutoValidatedتغییر کند.

رویداد BackgroundLayoutChanged : وقتی رخ می‌دهد که خاصیت BackgroundLayout تغییر کند.

رویداد ClientSizeChanged : وقتی رخ می‌دهد که خاصیت ClientSize تغییر کند.

رویداد ControlAdded : وقتی رخ می‌دهد که کنترلی اضافه شود.

رویداد ControlRemoved : وقتی رخ می‌دهد که کنترلی از روی فرم حذف شود.

رویداد HelpButtonClicked : وقتی که دکمه علامت ؟ روی فرم کلیک شود، این رویداد رخ می‌دهد.

رویداد MouseCaptureChanged : وقتی رخ می‌دهد که خاصیت MouseCaptureتغییر کند.

رویداد MouseDoubleClick : وقتی که دکمه ماوس کلیک مضاعف شود، این رویداد رخ می‌دهد.

رویداد PaddingChanged : هرگاه خاصیت Padding تغییر کند این رویداد رخ می‌دهد.

رویداد RoginChanged : هرگاه خاصیت Roginتغییر کند این رویداد رخ می‌دهد.

رویداد ResizeBegin : هرگاه تغییر اندازه فرم شروع شود، این رویداد رخ می‌دهد.

رویداد ResizeEnd : هرگاه تغییر اندازه فرم خاتمه یابد، این رویداد رخ می‌دهد.

رویداد Scroll : هرگاه فرم Scroll شود، این رویداد رخ می‌دهد.

Anonymous Methods

توابع بی نام[1]روشی برای الحاق یک تابع به نماینده[2]می‌باشد. یک تابع بی نام قطعه کدیست که بعنوان پارامتر برای نماینده استفاده می‌شود.

شکل تعریف نماینده که از تابع بی نام استفاده می‌کند، تغییر نمی‌کند. دو قطعه کد زیر دو ویرایش از یک کلاس را نشان می‌دهند. ویرایش اول تعریف و استفاده از تابع معمولی و کد زیرین استفاده از تابع بی نام را برای نماینده نشان می‌دهد.

---

[1] Anonymous Methods

[2] Delegate

```
class Program
{
    public static int Add20(int x)
    {
        return x + 20;
    }
    delegate int OtherDel(int InParam);
    static void Main()
    {
        OtherDel del = Add20;
        Console.WriteLine("{0}", del(5));
        Console.WriteLine("{0}", del(6));
    }
}
```

```
class Program
{
    delegate int OtherDel(int InParam);
    static void Main()
    {
        OtherDel del = delegate(int x)//  استفاده از تابع بی نام
        {
            return x + 20;
        };
        Console.WriteLine("{0}", del(5));
        Console.WriteLine("{0}", del(6));
    }
}
```

املای یک تابع بی نام شامل موارد زیر است:

- کلمه کلیدی نوع delegate
- لیست پارامترها (در صورتی که تابع هیچ پارامتری نداشته باشد، حذف می‌شود)
- قطعه دستوری یا همان کدهای مربوط به تابع

```
delegate ( Parameters ) { ImplementationCode }
```

نوع برگشتی

برای یک تابع بی نام بطور مشخص نوع برگشتی تعریف نمی‌شود. کد مربوط به تابع بی نام به قسمی‌باید باشد که نوع مقدار برگشتی آن با نوع برگشتی نماینده همخوانی داشته باشد. اگر نوع برگشتی نماینده از نوع void باشد کد اجرایی تابع بی نام نیز نمی‌تواند مقداری را برگرداند.

```
delegate int OtherDel(int InParam);
static void Main()
{
  OtherDel del = delegate(int x)
  {
    return x + 20 ;  // int نوع از برگشتی مقدار
  };
  ...
}
```

پارامترها

پارامترهای یک تابع بی نام باید از نظر تعداد پارامترها و نوع آنها با نماینده مربوطه همخوانی داشته باشند. می‌توان برای ساده سازی، پارامترهای یک تابع بی نام را در نظر نگرفت به شرطی که اولا لیست پارامترهای نماینده حاوی پارامتر out نباشد و ثانیا تابع بی نام هیچ پارامتری را مورد استفاده قرار نداده باشد. بعنوان مثال کد زیر نماینده را تعریف می‌کند که پارامتر out ندارد و تابعی بی نام نیز هیچ پارامتری را مورد استفاده قرار نمی‌دهد حال که هر دو شرط برقرار هستند می‌توان لیست پارامترها را در تابع بی نام نیاوریم.

```
delegate void SomeDel ( int X );
SomeDel SDel = delegate
{
  PrintMessage();
  Cleanup();
};
```

اگر لیست پارامترهای نماینده حاوی پارامتری از نوع params باشد، عبارت params از لیست پارامترهای تابع بی نام حذف می‌شود.

```
delegate void SomeDel( int X, params int[] Y);

SomeDel mDel = delegate (int X, int[] Y)
{
  ...
};
```

دسترسی به متغیرها

متغیرهای محلی قابل دسترسی در تابع بی نام هستند اما عکس این موضوع صادق نیست یعنی متغیر تعریف شده در یک تابع بی نام در خارج آن قابل دسترسی نیستند.

```
delegate void MyDel(int x);
int x = 5;

MyDel mDel = delegate(int y)
  {
    int z = x * 2;
    Console.WriteLine("{0}, {1}", y, z);
  };

Console.WriteLine("{0}, {1}", y, z);
                          //این خط از کد با خطای کامپایلر مواجه خواهد شد
```

- استفاده از توابع بی نام باعث کاهش کد نویسی می‌شود و شما دیگر مجبور به تعریف تابعی نیستید که فقط می‌خواهید آن را در نماینده استفاده کنید. این قضیه وقتی بیشتر مشهود می‌شود که نماینده را برای رویدادها[1]تعریف می‌کنید. این کار باعث کم شدن پیچیدگی کد می‌شود بخصوص زمانی که رویدادهای متعددی را تعریف می‌کنید.
- استفاده از تابع بی نام باعث سریعتر شدن اجرای کد نمی‌شود. کامپایلر همچنان یک تابع را تعریف می‌کند، تابعی که به طور خودکار برای آن نامی‌در نظر گرفته می‌شود که ما نیازی به دانستن آن نداریم.
- هیچگاه نمی‌توان در یک تابع بی نام از دستورات جهشی نظیر break، goto و یا continue به قسمی‌که کنترل جریان برنامه را به خارج از تابع بی نام منتقل کنند، استفاده کرد. عکس این موضوع هم صادق است یعنی نمی‌توان از خارج از یک تابع بی نام به داخل آن جهش داشت.
- کدهای ناامن[2]در تابع بی نام قابل دسترسی نیستند. پارامترهای out و ref که خارج از یک تابع بی نام تعریف شده اند نیز در داخل آن قابل دسترسی نیستند.

---

[1] Events

[2] Unsafe Block

- اگر چندین بار مجبور به نوشتن یک قطعه کد کاربردی مشابه هستید از تابع بی نام استفاده نکنید. در اینگونه موارد به جای تکرار آن قطعه کد مشابه در قالب تابع بی نام، نوشتن آن بصورت یک تابع ساده و ارجاع به آن توسط نام آن ارجحیت دارد.

Lambda Expressions

عبارات لامبدا[1] ساده شده دستور زبان متدهای بی نام[2] هستند. به عنوان مثال در برنامه زیر از یک متد بی‌نام که به یک نماینده[3] ارجاع داده شده است استفاده شده.

```
public delegate void MessageDelegate(string message);
public class Program
{
    public static void Main()
    {
        MessageDelegate ShowMessage = new MessageDelegate( delegate(string message)
            {
                Console.WriteLine(message);
            });
        ShowMessage("این یک متن آزمایشی می‌باشد");
    }
}
```

---

[1] Lambda expressions

[2] Anonymous Methods

[3] delegate

به وسیله عبارات لامبدا می‌توان الگوریتم قبل را به صورت ساده‌تری نوشت:

```
public delegate void MessageDelegate(string message);
public class Program
{
   public static void Main()
   {
      MessageDelegate ShowMessage = (message) => Console.WriteLine(message);
      ShowMessage("این یک متن آزمایشی می‌باشد");
   }
}
```

مقایسه متد بی نام[1] و عبارت لامبدا:

```
MyDel del = delegate(int x) { return x + 1; };        // متد بی نام
MyDel le1 = (x) => { return x + 1; };        // عبارت لامبدا
```

در عبارت لامبدا ابتدا پارامترها را بدون ذکر نوعشان می نویسیم و بعد از عملگر => استفاده می‌کنیم. سپس دستوراتی را که قرار است اجرا شوند را می نویسیم. نوع پارامترها به صورت خودکار به وسیله کامپایلر تشخیص داده می‌شود. البته امضاء عبارات لامبدا باید شبیه به امضاء نماینده[2] باشد. عبارات لامبدا دارای اشکال زیادی هستند. به عنوان مثال در مثال بالا از یک عبارت لامبدایی استفاده کرده ایم که دارای یک دستور ساده اجرایی است.

---

[1] Anonymous Method

[2] Delegated Signatures

- اگر نماینده برای عبارت لامبدا هیچ پارامتری نداشته باشد، همه کاری که لازم است انجام دهید این است که هیچ پارامتری در داخل عبارت لامبدا قرار ندهید.

```
MessageDelegate ShowMessage = () => Console.WriteLine("Hello");
```

- می‌توانید نوع پارامترهای عبارت لامبدا را نشان دهید.

```
MessageDelegate ShowMessage = (string message) => Console.WriteLine(message);
```

- اگر نوع یک پارامتر را مشخص کنید، باید نوع سایر پارامترها را نیز مشخص کنید. به عنوان مثال نمی‌توانید به صورت زیر عمل کنید:

```
MessageDelegate ShowMessage = (string message1, message2) => Console.WriteLine(message1);
```

- اگر یک عبارت لامبدا دارای ۲ یا تعداد بیشتری پارامتر باشد باید انها را داخل پرانتز قرار دهید.

```
SampleDelegate GetSum = (num1, num2) => num1 + num2;
```

- اگر عبارات لامبدای شما دارای چندین دستور اجرایی باشند می‌توانید آنها را داخل آکولاد قرار دهید. به عبارت لامبدایی که دارای آکولاد باشد دستور لامبدا می‌گویند.

```
MessageDelegate ShowMessage = (message) =>
{
    Console.WriteLine(message);
    Console.WriteLine("Some more message");
}
```

- در زیر مثالی از یک عبارت لامبدا که دارای مقدار برگشتی است نشان داده شده است:

```
SampleDelegate GetSquare = (number) => { return number * number; };
```

- هنگام استفاده از دستور return باید همیشه از دستورات لامبدایی استفاده کنید که دارای آکولاد می‌باشند. اگر یک عبارت لامبدا فقط دارای یک دستور return ساده باشد می‌توانید به سادگی آن را به عبارت لامبدا تبدیل کنید.

```
SampleDelegate GetSquare = (number) => (number * number);
```

- به این نکته توجه کنید که استفاده از پرانتز در کد بالا برای فهم بهتر آن است. اگر یک عبارت لامبدا دارای یک پارامتر ساده و یک دستور return است، می‌توانید برای سادگی بیشتر پرانتزها را حذف نمایید:

```
SampleDelegate GetSquare = number => number * number;
```

Enumeration and Iterators

تمامی‌کلاسهایی که به نحوی شامل یک کلکسیون[1] هستند دو رابط IEnumerable و IEnumerator را پیاده سازی می‌کنند.

وجود IEnumerable که توسط کلاسها پیاده سازی می‌شود به کلاس این امکان را می‌دهد که بصورت ضمنی و توکار بشود شیء را پیمایش کرد. دقیقا به همین دلیل می‌توان با استفاده از حلقه foreach یک آرایه را پیمایش کرد، چون که رابط IEnumerable توسط کلاس System.Array پیاده سازی می‌شود.

رابط IEnumerator در سطح پایین تری از یک IEnumerable قرار دارد. با استفاده از این رابط می‌توان در هرجای بدنه متد اشیایی را برگشت داد بدون اینکه مجبور باشید ابتدا نتایج را مثلا در یک آرایه بریزید و بعد آن آرایه را برگشت دهید.

---

[1] Collection

رابط IEnumerator

یک شمارنده[1] رابط IEnumerator را پیاده سازی می‌کند که دارای دو متد MoveNext()، Reset() و یک خاصیت به نام Current می‌باشد. خاصیت Current عنصر جاری یک مجموعه را بر می‌گرداند. این خاصیت یک خاصیت فقط خواندنی[2] است و چیزی که برمی‌گرداند از نوع object می‌باشد.

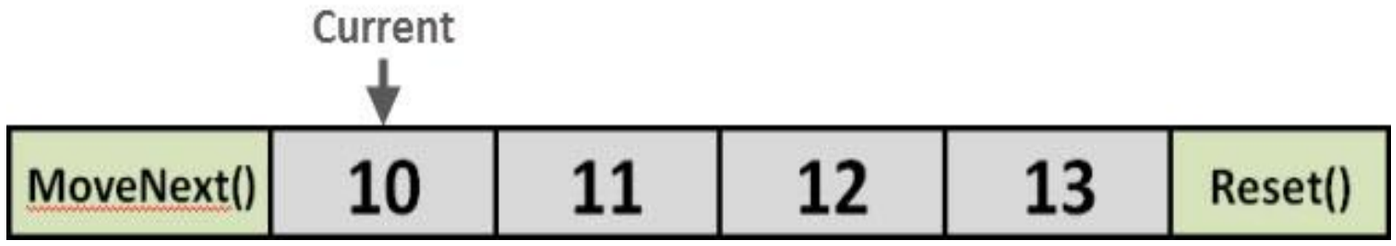
متد MoveNext() متدی است که، مکان شمارنده را از یک آیتم در مجموعه به آیتمی‌دیگر منتقل می‌کند. این متد یک مقدار بولی را بر می‌گرداند که نشان می‌دهد که آیا مکان دیگری برای خواندن در دسترس است یا به انتهای مجموعه رسیده است. اگر مکان جدیدی وجود داشته باشد مقدار true و در غیر اینصورت مقدار false را بر می‌گرداند. مکان اولیه شمارنده قبل از اولین آیتم مجموعه است، بنابراین MoveNext() باید قبل از اولین دسترسی خاصیت Current فراخوانی شود.

---

[1] Enumerator

[2] read-only

متد Reset() متدی برای برگرداندن شمارنده به موقعیت اولیه خود قبل از جابجایی است .به تعبیری دیگر، موقعیت اولیه مجموعه را برمی‌گرداند . با در اختیار داشتن یک شمارنده[1]شما قادر خواهید بود که حلقه foreach را شبیه سازی کرده و عناصر یک مجموعه را با استفاده ازمتد MoveNext() و خاصیت Current پیمایش کنید. برای درک بهتر عملکرد دو متد و خاصیت مذکور به شکل و کد زیر توجه کنید




```
int[] numbers = { 10, 11, 12, 13 };
IEnumerator IEnumerator1 = numbers.GetEnumerator();
IEnumerator1.MoveNext();
int i = (int)IEnumerator1.Current;

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

```
10
```

[1] enumerator

همانطور که در کد بالا مشاهده می‌کنید متد GetEnumerator() آرایه numbers را به نوع شمارش پذیر تبدیل می‌کند، سپس با فراخوانی متد MoveNext() عدد ۱۰ که اولین عضو آرایه است به عنوان عنصر جاری (Current) برگردانده می‌شود. حال فرض کنید که شما می‌خواهید عدد ۱۲ را چاپ کنید، برای این کار باید متد MoveNext() را سه بار فراخوانی کنید:

```
int[] numbers = { 10, 11, 12, 13 };

IEnumerator IEnumerator1 = numbers.GetEnumerator();
IEnumerator1.MoveNext();
IEnumerator1.MoveNext();
IEnumerator1.MoveNext();
int i = (int)IEnumerator1.Current;
Console.WriteLine(i.ToString());
```

```
12
```

آرایه ها از انواع قابل شمارش[1] هستند، بنابراین کد زیر روش دستی کاری است که حلقه foreach به صورت خودکار انجام می‌دهد. در حقیقت کامپایلر سی‌شارپ کدی شبیه به کد زیر را در هنگام نوشتن دستور foreach تولید می‌کند:

```
public class Program
{
    public static void Main()
    {
        int[] numbers = { 10, 11, 12, 13 };
        IEnumerator IEnumerator1 = numbers.GetEnumerator();
        while (IEnumerator1.MoveNext())
        {
            int i = (int)IEnumerator1.Current;
            Console.WriteLine("{0}", i);
        }
    }
}
```

```
10
11
12
13
```

[1] enumerable

رابط IEnumerable

یک کلاس قابل شمارش کلاسی[1] است که رابط IEnumerable را پیاده سازی کند. رابط IEnumerable فقط یک عضو دارد و آن عضو هم متد GetEnumerator() می‌باشد و پارامتر برگشتی این متد از نوع همان رابط IEnumberator است. این رابط در واقع کلاس ما را قابل پیمایش می‌کند تا بتوانیم حلقه foreach را در مورد کلاسمان بکار ببریم. فرم کلی بصورت زیر است:

```
using System.Collections;
class MyClass : IEnumerable
{
    public IEnumerator GetEnumerator
    {
        ...
    }
    ...
}
```

---

[1] enumerable

اکنون یک مثال را با هم مرور می‌کنیم. فرض کنید یک کلاس به نام ColorEnumerator که رابط IEnumerator را پیاده سازی می‌کند موجود است. این کلاس یک رشته از رنگ‌ها را در بر می‌گیرد، و چون رابط IEnumerator را پیاده سازی می‌کند، پس قابل پیمایش می‌شود:

```
using System;
using System.Collections;
class ColorEnumerator : IEnumerator
{
    string[] Colors;
    int Position = -1;
    public ColorEnumerator(string[] theColors) // Constructor
    {
        Colors = new string[theColors.Length];
        for (int i = 0; i < theColors.Length; i++)
            Colors[i] = theColors[i];
    }
    public object Current// Implement Current.
    {
        get
        {
            return Colors[Position];
        }
    }
```

```
    public bool MoveNext()// Implement MoveNext.
    {
        if (Position < Colors.Length - 1)
        {
            Position++;
            return true;
        }
        else
            return false;
    }
    public void Reset()// Implement Reset.
    {
        Position = -1;
    }
}
```

اکنون کلاسی دیگر ایجاد می‌کنیم که رابط IEnumerable را پیاده سازی می‌کند، این کلاس در متد GetEnumerator() پارامتری از نوع کلاس بالا برمی‌گرداند مطابق شکل زیر:

```
class MyColors : IEnumerable
{
    string[] Colors = { "Red", "Yellow", "Blue" };
    public IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(Colors);
    }
}
```

حال در برنامه براحتی می‌توانیم از حلقه foreach برای پیمایش اعضای آن استفاده کنیم.

به کد زیر توجه کنید:

```
class Program
{
    static void Main()
    {
        MyColors MC = new MyColors();
        foreach (string color in MC)
            Console.WriteLine(color);
    }
}
```

```
Red
Yellow
Blue
```

Iterators

هنگامی‌که می‌خواهید در متدهای خود مقداری (از هر نوع datatype دلخواه) را return نمایید، در حالت عادی قادر خواهید بود که فقط از یک return در بدنه متد خود استفاده نمایید:

```
public int Sum(int a, int b) { return a + b; }
```

متدهای تکرار شونده[1] در داخل یک کلکسیون به صورت دلخواه iterate کرده یا به اصلاح پیمایش می‌کنند. این متدها از کلمه کلیدی Yield در هنگام return کردن مقادیر استفاده می‌کنند. (در سی‌شارپ از Yield return و در ویزوال بیسیک از Yield استفاده می‌شود) به عبارت دیگر یک متد با خروجی از نوع قابل پیمایش (مانند IEnumerable)، با استفاده از چند yield return، دارای قابلیت پیمایش و بازگرداندن چندین مقدار به جای یک مقدار واحد می‌گردد.

---

[1] Iterator method

برای درک بهتر مسئله از مثالی در ادامه توضیحات آورده می‌شود. متد پیمایش شونده[1] زیر را در نظر بگیرید که خروجی IEnumerable دارد:

```
public static IEnumerable SomeNumbers()
{
     yield return 3;
     yield return 5;
     yield return 8;
}
```

[1] Iterate method

برای استفاده از مقادیر بازگشتی متد بالا از حلقه foreach زیر استفاده می‌نماییم:

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}
```

حلقه foreach فوق ، در پایان اولین پیمایش، عدد ۳ را باز گردانده و مکان این return را حفظ می‌کند. در چرخه بعدی عدد ۵ را باز می‌گرداند و این نقطه را نیز نگه می‌دارد و در چرخه پایانی عدد ۸ را برگردانده و سپس حلقه با رسیدن به نقطه پایانی متد، خاتمه می‌یابد.

برای خاتمه پیمایش در متدهای پیمایش شونده، می‌توانید از foreach استفاده کنید و یا اینکه عبارت yield break را بعد از تمامی yield returnها به کار گیرید:

```
public static IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
    yeild break;
}
```

- در هنگام ایجاد متدهای پیمایش شونده، نوع مقادیر خروجی متد باید یکی از انواع IEnumerable, IEnumerator, IEnumerable<T>, و یا IEnumerator<T>. باشد.
- در هنگام فراخوانی[1] نمی‌توانید از پارامترهای ref و out استفاده نمایید.
- در متدهای بی نام[2] و بلاک های ناامن[3] نمی‌توانید از yield return (yield در VB) استفاده نمایید.
- نمی‌توانید از yield return در بلوکهای try-catch استفاده کنید. اما می‌توانید در قسمت try بلوک try-finally استفاده نمایید.
- از yield break می‌توانید در بلوک try و یا بلوک catch استفاده نمایید، اما در بلوک finally خیر.
- هنگام بروز خطا در foreach هایی که خارج از متدهای پیمایش شونده استفاده می‌شوند، بلوک finally داخل این متدها اجرا می‌گردد.

---

[1] declare

[2] Anonymous method

[3] Unsafe Block

Nullable Types

همانطور که در بخش مربوط به ساختار ها بیان شد، مقدار پیش فرض یک ساختار شکل های مختلفی از [1] می‌باشد. این یکی از تفاوت های موجود بین انواع ارجاعی و انواع مقداری بحساب می‌آید. مقدار پیش فرض یک نوع ارجاعی برابر با null است.

چنانچه شما در برنامه سی‌شارپ، خودتان منبع داده های برنامه را مدیریت و کنترل کنید، مثلا از فایل‌هایی برای نگهداری داده‌ها استفاده نمایید، مقادیر پیش فرض ساختارها برای شما بخوبی عمل خواهند کرد. اما در واقعیت معمولا برای اینکار ها از بانک‌های اطلاعاتی استفاده می‌شود که سیستم‌های نوع[2] مخصوص خودشان را دارند.

در کار با سیستم‌های نوع بانک‌های اطلاعاتی به این مسئله باید توجه کنید که نیازی نیست نگاشتی یک به یک را بین انواع موجود در سی‌شارپ و بانک اطلاعاتی داشته باشید.

---

[1] Struct

[2] type system

یکی از تفاوت های اصلی بین آنها این است که انواع در بانک اطلاعاتی می‌توانند مقدار null را بپذیرند. بانک اطلاعاتی هیچ شناختی نسبت به انواع ارجاعی و مقداری که جزئی از مفاهیم موجود در سی‌شارپ هستند ندارد و این انواع برای آن بی معنی هستند. این باعث می‌شود که انواع مقداری سی‌شارپ در نظیر int، decimal, و datetime در پایگاه داده می‌توانند مقدار null را بپذیرند.

حال که می‌دانیم انواع در بانک اطلاعاتی می‌توانند مقدار null را بپذیرند ولی در سی‌شارپ قادر به این کار نیستند، باید راهی پیدا کنیم که بتوانیم مقادیر null را برای انواع در سی‌شارپ نیز تعریف کنیم. روش‌هایی که معمولا برای اینکار استفاده می‌شوند، از برنامه‌ای به برنامه دیگری فرق می‌کنند. ممکن است یک روش در یک برنامه بخوبی کار کند، اما استفاده از آن در برنامه‌ای دیگر ناکار آمد باشد. برخی اوقات نیز ممکن است راهی برای این کار وجود نداشته باشد. برای برطرف کردن این مشکل از سی‌شارپ ۲,۰ به بعد نوعی جدید به این به این زبان اضافه شد که توسط آن براحتی می‌شود با مقادیر null کار نمود.

تعریف انواع null پذیر

برای آنکه یک نوع مقداری را از نوع نال پذیر[1] تعریف کنیم، علامت سوالی را بعد از نام نوع اضافه می‌کنیم. در ادامه با هم می‌بینیم که چگونه می‌توان متغیری از نوع DateTime تعریف نمود که نال پذیر نیز باشد.

```
DateTime? startDate;
```

همانطور که می‌دانید بطور معمول نوع DateTime نمی‌تواند یک مقدار null را در خود نگهداری کند. اما نحوه تعریف ما در مثال بالا منجر می‌شود که این نوع قادر به انجام این کار نیز شود.

در واقع با این تعریف نوع متغیر startDate بعنوان یک DateTime با قابلیت پذیرش مقدار null می‌شود.

[1] Nullable Type

حال شما می‌توانید هر مقداری را که سازگار با این نوع باشد به آن دهید. مثلا می‌توانید تاریخ فعلی را در آن ذخیره کنید.

```
startDate = DateTime.Now;
```

و یا بصورت زیر مقدار null را به آن نسبت دهید:

```
startDate = null;
```

نکته: نحوه تعریف یک نوع نال پذیر را با مقدار دهی اولیه آن می‌بینیم:

```
int? unitsInStock = 5;
```

همانند مثال قبلی به متغیر unitsInStock نیز می‌توانیم مقدار null را نسبت دهیم.

نحوه کار کردن با انواع null پذیر:

معمولا وقتی با انواع نال پذیر کار می‌کنیم، می‌خواهیم بدانیم مقدار آنها null است یا خیر؟ مثال زیر چگونگی چک کردن مقدار یک نوع نال پذیر را به ما نشان می‌دهد:

```
bool isNull = startDate == null;
Console.WriteLine("isNull: " + isNull);
```

همانطور که در مثال بالا می‌بینید شما تنها نیاز به عملگر برابری برای بررسی مقدار null دارید. البته اینکار را با استفاده از عبارات شرطی نیز می‌توانید انجام دهید:

```
int availableUnits;

    if (unitsInStock == null)
        availableUnits = 0;
    else
        availableUnits = (int)unitsInStock;
```

در مثال بالا به عملگر تبدیل نوع[1] در عبارت else توجه کنید. در هنگام نسبت دادن مقدار یک نوع نال پذیر به انواع غیر نال پذیر باید بصورت صریح این تبدیل نوع را انجام داد.

نکته: برای ساده ترشدن برنامه، می‌توان به‌جای نوشتن چنین عبارات شرطی ای که باعث پیچیدگی می‌شوند، از عملگر ?? استفاده نمود. مثلا عبارت شرطی بالا را می‌توان بصورت زیر نیز نوشت:

```
int availableUnits = unitsInStock ?? 0;
```

معنی این عملگر به این صورت خواهد بود که چنانچه مقدار unitsInStock برابر null بود، عبارت موجود در سمت راست آن(۰) را لحاظ کن.

---

[1] cast

Operator Overloading

جالب است بدانید؛ می‌توانید اکثر اپراتورهای موجود و داخلی در سی‌شارپ را دوباره تعریف[۱] کنید. بنابراین برنامه نویس می‌تواند از اپراتورهایی با نوع تعریف شده توسط کاربر نیز استفاده کند. اپراتورهای باز تعریف شده[۲] عملکردهایی هستند با نام‌های خاص که کلمه ی کلیدی operator با نمادی برای اپراتور تعریف شده دنبال می‌شود. مانند هر عملکرد دیگری یک اپراتور بازتعریف شده دارای یک نوع بازگشتی و یک لیست پارامتر می‌باشد.

---

[۱] Overload

[۲] Operator Overloading

برای مثال به عملکرد زیر دقت کنید.

```
public static Box operator +(Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

عملکرد بالا اپراتور جمع (+) را برای یک گروه Box تعریف شده توسط یوزر، اجرا می‌کند.

اجرای بازتعریف کردن کردن اپراتور

برنامه ی زیر اجرای کامل را نشان می‌دهد.

```
namespace Operator_Overloading
{
    public class Box
    {
            private double length;      // Length of a box
            private double breadth;     // Breadth of a box
            private double height;      // Height of a box

            public double getVolume()
            {
               return length* breadth * height;
            }
            public void setLength(double len)
            {
               length = len;
            }
```

```
    public void setBreadth(double bre)
    {
        breadth = bre;
    }
    public void setHeight(double hei)
    {
        height = hei;
    }
    // Overload + operator to add two Box objects.
    public static Box operator +(Box b, Box c)
    {
        Box box = new Box();
        box.length = b.length + c.length;
        box.breadth = b.breadth + c.breadth;
        box.height = b.height + c.height;
        return box;
    }
}
```

```
class Tester
{
    static void Main(string[] args)
    {
        Box Box1 = new Box();// Declare Box1 of type Box
        Box Box2 = new Box();// Declare Box2 of type Box
        Box Box3 = new Box();// Declare Box3 of type Box
        double volume = 0.0;// Store the volume of a box here
        // box 1 specification
        Box1.setLength(6.0);
        Box1.setBreadth(7.0);
        Box1.setHeight(5.0);
        // box 2 specification
        Box2.setLength(12.0);
        Box2.setBreadth(13.0);
        Box2.setHeight(10.0);
        // volume of box 1
        volume = Box1.getVolume();
        Console.WriteLine("Volume of Box1 : {0}", volume);
        // volume of box 2
```

```
            volume = Box2.getVolume();
            Console.WriteLine("Volume of Box2 : {0}", volume);
            // Add two object as follows:
            Box3 = Box1 + Box2;
            // volume of box 3
            volume = Box3.getVolume();
            Console.WriteLine("Volume of Box3 : {0}", volume);
            Console.ReadKey();
        }
    }
}
```

زمانی که برنامه ی بالا کامپایل شده و اجرا می‌شود، نتایج زیر را به دنبال دارد.

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

جدول زیر توانایی باز تعریف شدن اپراتورها را در سی‌شارپ توضیح می‌دهد.

Operators	Description
+, -, !, ~, ++, --	این عملگرهای یکانی یک عملوند می‌گیرند ظو می‌توانند overload شوند.
+, -, *, /, %	این عملگرهای دودویی یک عملوند می‌گیرند و می‌توانند overload شوند.
==, !=, <, >, <=, >=	این عملگر های مقایسه ای می‌توانند overload شوند.
&&, ||	عملگر های منطقی شرطی نمی‌توانند به طور مستقیم overload شوند.
+=, -=, *=, /=, %=	عملگرهای تخصیص نمی‌توانند overload شوند.
=, ., ?:, ->, new, is, sizeof, typeof	این عملگرهای نمی‌توانند overload شوند.

هنگامی‌که یک عملگر باز تعریف می‌شود، معنای واقعی خودش را از دست نمی‌دهد. بلکه فقط کاربرد آن به یک کلاس افزوده می‌شود. بنابراین (به‌عنوان مثال) باز تعریف کردن عملگر + برای افزودن یک شیء به انتهای لیست پیوندی، دلیل نمی‌شود که عملکرد آن operator برای جمع کردن دو عدد صحیح تغییر کند.

مزیت اصلی باز تعریف نمودن عملگرها این است که به شما اجازه می‌دهد به‌طور یکپارچه، یک کلاس جدید را در محیط برنامه‌نویسی خود، ادغام کنید. این ویژگی که به آن type extensibility می‌گویند، یکی از بخش‌های مهم یک زبان برنامه‌نویسی شی‌گرا مثل سی‌شارپ است. هنگامی‌که عملگرها برای یک کلاس تعریف می‌شوند، می‌توانید آن را روی اشیای کلاس مربوطه، اعمال کنید. این نکته قابل ذکر است که باز تعریف نمودن عملگرها یکی از قدرمندترین ویژگی‌های سی‌شارپ است.

اصول باز تعریف نمودن عملگرها

باز تعریف نمودن عملگرها شباهت زیادی با باز تعریف نمودن متدها[۱]دارد. دو حالت از باز تعریف نمودن متدها وجود دارد: عملگرهای تکی[۲]و عملگرهای دوتایی.[۳]فرم کلی هرکدام را در زیر می‌بینید:

```
// General form for overloading a unary operator
public static ret-type operator op(param-type operand)
{
        // operations
}
// General form for overloading a binary operator
public static ret-type operator op(param-type1 operand1, param-type1 operand2)
{
        // operations
}
```

---

۱ Method overloading

۲ unary operators

۳ binary operators

در این‌جا، عملگری که آن را بازتعریف می‌کنید، مثل + یا / ، جایگزین op می‌شود. ret-type مشخص کننده‌ی نوع مقداری است که برگشت داده خواهد شد. اگرچه نوع بازگشتی[1] می‌تواند از هر نوعی باشد اما اغلب از نوع همان کلاسی است که عملوند در آن سربار[2] می‌شود. این ارتباط (یکسان بودن نوع بازگشتی با جنس کلاس) باعث راحتی استفاده از عمگرهای بازتعریف شده می‌شود. برای عملگرهای تکی ، عملوند در قسمت operand قرار می‌گیرد. برای عملگرهای دوتایی ، عملوندها در قسمت operand۱ و operand۲ قرار خواهد گرفت.

---

[1] Return Type

[2] Overload

- متدهای بازتعریف شده باید هم public و هم static باشند.
- در عملگرهای تکی ، نوع عملوند باید با نوع کلاسی که operator در آن تعریف می‌شود، یکسان باشد. بنابراین نمی‌توانید عملوندهای سی‌شارپ را برای اشیایی که خودتان نساخته‌اید، تعریف کنید. برای مثال، نمی‌توانید مجدداً عملگر + را برای int و string تعریف کنید.
- پارامترهای عملوند نباید از ref وout استفاده کنند.

Extension Methods

یکی از قابلیتهای جدید و جالب توجه سی شارپ ۳ متدهای تعمیم یافته[۱]هستند. یک متد تعمیم یافته به یک متد استاتیک گفته می‌شود که در یک کلاس از نوع استاتیک وجود دارد و شما می‌توانید بر خلاف متدهای استاتیک عادی، روی نمونه‌های[۲]کلاس، مثل یک متد عادی برای آن نمونه از کلاس استفاده کنید اما قادر نخواهید بود که همان متد را در سطح کلاس استفاده نمایید. برای درک تفاوت متدهای سطح کلاس و متدهای استاتیک عادی به مثال توجه نمایید.

---

[۱] Extension Methods

[۲] Instance

به عنوان مثال متد ToUpper یک متد عادی تعریف شده در سطح کلاس string است که فقط روی نمونه های کلاس می‌توان از آن استفاده کرد مثلا:

```
String  name = "ali" ;
Console.WriteLine(name.ToUpper());
```

بعد از اجرای کد بالا کلمه "Ali" به کاربر نشان داده خواهد شد. حال اگر بخواهیم همین متد را روی کلاس استفاده کنیم یعنی مثلا string.ToUpper() با خطای کامپایلر مواجه خواهیم شد.

بالعکس اگر بخواهیم متدهای استاتیک تعریف شده داخل کلاس را روی نمونه های کلاس استفاده کنیم، با خطا مواجه می‌شویم. به عنوان مثال متد Format در سطح کلاس string تعریف شده، یعنی اگر بنویسیم: name.Format(.....) با خطای کامپایلر روبرو خواهیم شد در صورتی که نحوه استفاده صحیح از این متد string.Format(...) می‌باشد.

اما متدهای تعمیم یافته علاوه بر اینکه متد استاتیک هستند، به شما اجازه می‌دهند که آنها را در روی نمونه های گرفته شده از کلاس استفاده کنید، نه روی خود کلاس.

تعریف و فراخوانی یک متد تعمیم یافته

به نکات زیر توجه کنید:

- در تعریف یک متد تعمیم یافته باید دقت کنیم که حتما آنرا داخل یک کلاس استاتیک تعریف نماییم.
- اولین آرگومان متد تعریف شده باید با کلیدواژه this تعریف گردد.
- متد تعمیم یافته باید از نوع استاتیک باشد.

مثال:

```
namespace csharp.Utilities
{
    public static class StringConversions
    {
        public static double ToDouble(this string s)
        {
            return Double.Parse(s);
        }
        Public static bool ToBool(this string s)
        {
            return Boolean.Parse(s);
        }
    }
}
```

حال از متدهای تعریف شده استفاده می‌کنیم

```
using csharp.Utilities
double pi = "3.1415926335".ToDouble();
Console.WriteLine(pi);
```

اگر بخواهیم میان متدهای تعمیم یافته و متدهای عادی اولویت قرار دهیم، متدهای عادی اولویت بیشتر را به خود اختصاص می‌دهند، به دلیل اینکه متدهای تعمیم یافته دارای محدودیت عملکرد هستند. اما به طور کلی متدهای بسیار مفیدی بوده و خصوصا در بخش‌هایی که نیاز به بسط دادن یک کلاس داریم اما به دلایل مختلفی از قبیل عدم دسترسی به سورس کلاس و یا مهر و موم[1]بودن آن نمی‌توانیم این کار را انجام دهیم، بسیار مفید خواهند بود.

[1] sealed

جدول برخی از متدهای تعمیم یافته

Object extension:

Chaining	Value Comparison	Casting	Reflection
Chain	Between	As<T>	GetCustomAttribute
Serialization	In	IsAssignableFrom	GetCustomAttributeDescription
SerializeBinary	InRange	Convert	GetCustomAttributes
SerializeXml	IsDBNull	ChangeType	GetField
Cloning	IsDefault	GetTypeCode	GetFields
DeepClone	IsNotNull	To<T>	GetFieldValue
ShallowCopy	IsNull	To[ValueType]	GetMethod
Utility	NotIn	To[ValueType]OrDefault	GetMethods
Coalesce	ReferenceEquals	ToNullable[ValueType]	GetProperties
CoalesceOrDefault	Type Comparison	ToNullable[ValueType]OrDefault	GetProperty
NullIf	IsArray		GetPropertyValue
NullIfEquals	IsClass		InvokeMethod
NullIfEqualsAny	IsEnum		IsAttributeDefined
GetValueOrDefault	IsSubclassOf		SetFieldValue
	IsTypeOf		SetPropertyValue
	IsTypeOrInheritOf		

Database extension:

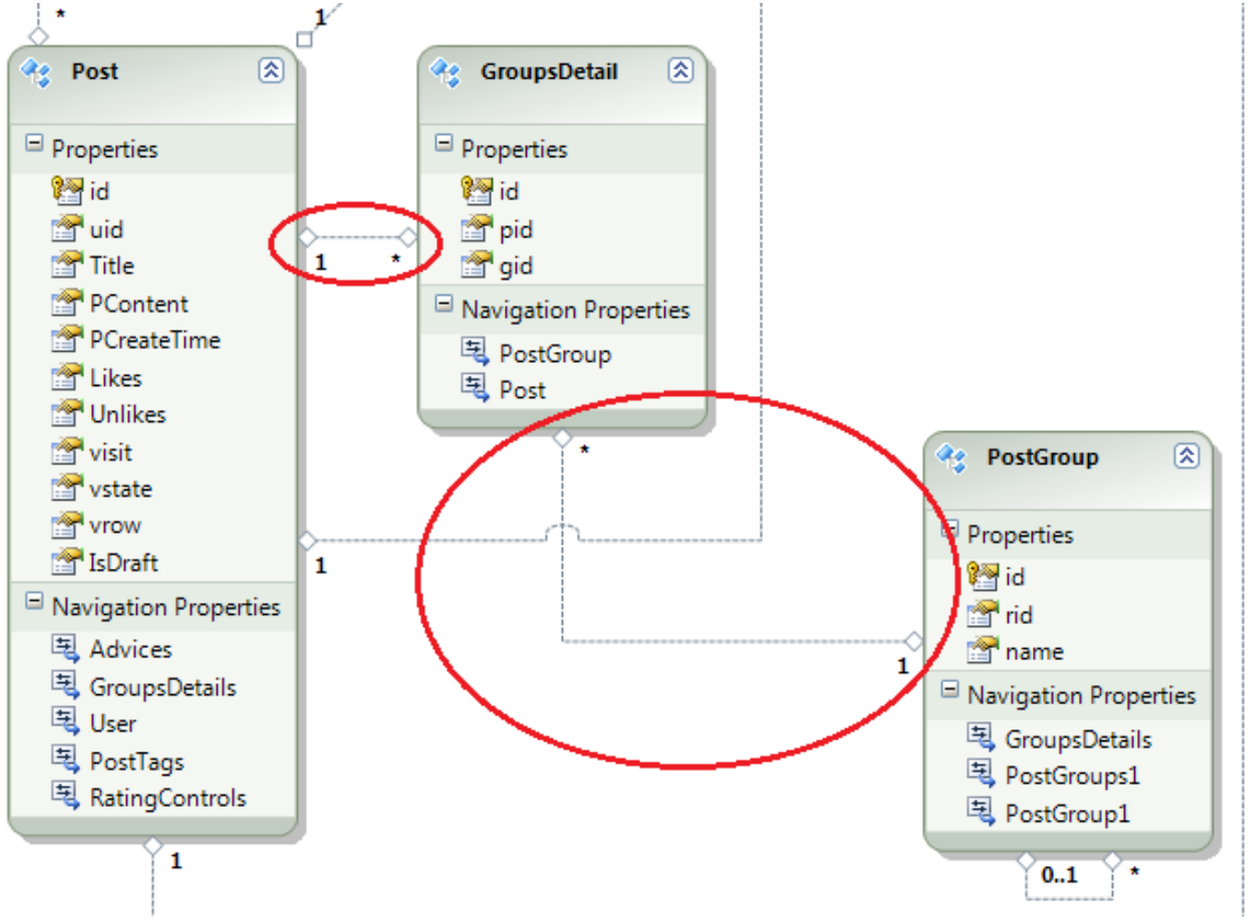
IDataReader	IDbConnection	DbCommand
ContainsColumn	EnsureOpen	ExecuteEntity
GetColumnNames	DbConnection	ExecuteEntities
Read All	ExecuteEntity	ExecuteExpandoObject
ForEach	ExecuteEntities	ExecuteExpandoObjects
ToDataTable	ExecuteExpandoObject	ExecuteScalarAs<T>
ToEntities	ExecuteExpandoObjects	ExecuteScalarAsOrDefault<T>
ToExpandoObjects	ExecuteNonQuery	ExecuteScalarTo<T>
Read Result	ExecuteReader	ExecuteScalarToOrDefault<T>
IsDBNull	ExecuteScalar	SqlCommand
GetValueAs<T>	ExecuteScalarAs<T>	ExecuteDataSet
GetValueAsOrDefault<T>	ExecuteScalarAsOrDefault<T>	ExecuteDataTable
GetValueTo<T>	ExecuteScalarTo<T>	SqlParameterCollection
GetValueToOrDefault<T>	ExecuteScalarToOrDefault<T>	AddRangeWithValue
ToEntity	SqlConnection	
ToExpando	ExecuteDataSet	
SqlBulkCopy	ExecuteDataTable	
GetSqlConnection	ConnectionState	
GetTransaction	In	
	NotIn	

String extension:

Check	Utility	Extract	System.Char
Contains	Br2Nl	ExtractWhere	ConvertToUTF32
ContainsAll	ConcatWith	ExtractLetter	GetNumericValue
ContainsAny	EscapeXml	ExtractNumber	GetUnicodeCategory
In	FormatWith	Encoding	IsControl
IsAlpha	IfEmpty	EncodeBase64	IsDigit
IsAlphaNumeric	Left	DecodeBase64	IsHighSurrogate
IsEmpty	LeftSafe	Encrypting	IsLetter
IsLike	Nl2Br	EncryptRSA	IsLetterOrDigit
IsNotEmpty	NullIfEmpty	DecryptRSA	IsLower
IsNotNull	Repeat	ToSecureString	IsLowSurrogate
IsNotNullOrEmpty	ReplaceByEmpty	Serialization	IsNumber
IsNull	Reverse	DeserializeBinary	IsPunctuation
IsNullOrEmpty	Right	DeserializeXml	IsSeparator
IsNumeric	RightSafe	RegularExpressions.Regex	IsSurrogate
NotIn	SaveAs	IsMatch	IsSurrogatePair
ToObject	Split	Match	IsSymbole
ToByteArray	Transform	Matches	IsUpper
ToDirectoryInfo	ToPlural	System.Web.HttpUtility	IsWhiteSpace
ToEnum	ToTitleCase	HtmlAttributeEncode	System.String
ToFileInfo	Truncate	HtmlDecode	CompareOrdinal
ToMemoryStream	RegexPattern	HtmlEncode	Concat
ToXDocument	IsValidEmail	JavaScriptStringEncode	Copy
ToXmlDocument	IsValidIP	ParseQueryString	Format
GetString	Remove	UrlDecode	Intern
GetAfter	RemoveDiacritics	UrlDecodeToBytes	IsInterned
GetBefore	RemoveLetter	UrlEncode	IsNullOrWhiteSpace
GetBetween	RemoveNumber	UrlEncodeToBytes	Join
	RemoveWhere	UrlPathEncode	

Anonymous Types

فرض کنید ساختار زیر را در مدل ساخته شده به وسیله‌ی Entity framework در پروژه‌ی خود داریم.



جدول Post با جدول GroupsDetail ارتباط یک به چند و در مقابل ان جدول GroupsDetail با جدول PostGroup ارتباط چند به یک دارد. به زبان ساده تعدادی گروه بندی برای مطالب وجود دارد (در جدول PostGroup) و می‌توان برای هر مطلب تعدادی از گروه‌ها را در جدول GroupsDetail مشخص نمود.

حال فرض کنید بخواهیم لیستی از عنوان مطالب موجود به همراه [نام] گروه‌های هر مطلب را داشته باشیم . به قطعه کد ساده‌ی زیر توجه فرمایید:

```
var context = new Models.EntitiesConnection();
var query = context.Posts.Select(pst => new {

    id = pst.id,
    Title = pst.Title,
    GNames = pst.GroupsDetails.Select
         (grd => new { Name = grd.PostGroup.name })

}).OrderByDescending(c => c.id).ToList();
```

همانطور که شاهد هستید در قطعه کد بالا توسط خواص راهبری[1] به صورت مستقیم نام گروه‌های ثبت شده برای هر مطلب در جدول GroupsDetail را از جدول PostGroup استخراج نمودیم.

نتیجه‌ی این query چه خواهد بود؟

کاملا واضح است که تعداد دلخواهی از فیلدها برای واکشی مشخص شده است؛ پس در نتیجه نوع داده ای که توسط این query بازگشت داده خواهد شد یک لیست از یک نوع بی نام می‌باشد.

چگونه از نتیجه‌ی بازگشتی این query در صورت ارسال آن به عنوان یک پارامتر به یک تابع می‌توان استفاده کرد؟

اگر ما تمامی فیلدهای جدول Post را واکشی کنید مقدار بازگشتی یک لیست از نوع Post خواهد بود که به راحتی قابل استفاده می‌باشد.

---

[1] Navigation Properties

مثال

```
public bool myfunc(List<Post> query)
{
    foreach (var item in query)
    {
            .....  string title = item.Title;
    }
...return true;
}
var context = new Models.EntitiesConnection();
var queryx = context.Posts.ToList();
.... myfunc(queryx );
```

اکنون با لیستی از نوع بی نام رو به رو می‌باشید. چند راه مختلف برای دسترسی به این گونه از مقادیر بازگشتی وجود دارد .

۱.استفاده از Reflection برای دسترسی به فیلدهای مشخص شده.

۲. تعریف یک مدل کامل بر اساس فیلدهای مشخص شده‌ی بازگشتی و ارسال یک لیست از نوع تعریف شده به تابع

۳.استفاده از یکی از روش‌های خلاقانه‌ی تبدیل نوع Anonymousها .

به کد زیر توجه فرمایید :

```
public static List<T> CreateGenericListFromAnonymous<T>(object obj, T example)
{
    return (List<T>)obj;
}
public static IEnumerable<T> CreateEmptyAnonymousIEnumerable<T>(T example)
{
    return new List<T>();
}
public bool myfunc(object query)
{
    var cquery = CreateGenericListFromAnonymous(query, new {
                            id = 0,
                            Title = string.Empty,
                            GNames = CreateEmptyAnonymousIEnumerable
                                        (new { Name = string.Empty })
                        });
    foreach (var item in cquery)
    {
        string title = item.Title;
        foreach (var gname in item.GNames)
            string gn = gname.Name;
    }
    return false;
}
```

```
var context = new Models.EntitiesConnection();
var query = context.Posts.Select(pst => new
{
    id = pst.id,
    Title = pst.Title,
    GNames = pst.GroupsDetails.Select
   (grd => new { Name = grd.PostGroup.name })
}).OrderByDescending(c => c.id);
            myfunc(query.ToList());
```

در کد بالا به صورت تو در تو از انواع بی نام استفاده شده است؛ تا مطلب کاملا روشن شود . بدین معنی که یک نوع بی نام که یکی از فیلدهای آن یک لیست از یک نوع بی نام دیگر است.

تابع CreateGenericListFromAnonymous یک object را گرفته و آن را به یک لیست تبدیل می‌کند (بر اساس نوعی که به صورت inline برای آن مشخص شده است)

تابع CreateEmptyAnonymousIEnumerable یک لیست از نوع IEnumerable را بر اساس نوعی که به صورت inline مشخص نموده ایم برمی‌گرداند.

Dynamic Binding

نوع داده پویا[1] به این معنی است که نوع شیء ، اعضا و پارامتر هایی که به آن ارسال می‌شوند تا زمان اجرا نامشخص می‌باشند. با تعریف یک شیء از نوع پویا به کامپایلر می‌گویید که کنترل نوع را در زمان کامپایل انجام ندهد به این ترتیب شما در زمان اجرا می‌توانید هر نوع مقداری را به شیء نسبت دهید. به این ترتیب شما قادر هستید به یک شیء از نوع داینامیک مقادیری از انواع مختلف بدهید.

```
dynamic d = "test";
Console.WriteLine(d.GetType());
// Prints "System.String".

d = 100;
Console.WriteLine(d.GetType());
// Prints "System.Int32".
```

---

[1] Dynamic

یکی از کاربردهای این نوع داده در زمان کار با COM API و یا زبان‌های داینامیکی مانند IronPython می‌باشد به این صورت که با نوع داده داینامیک دیگر نیازی نیست نگران این موضوع باشید که این زبان ها و یا API داده را با چه نوعی به شما باز می‌گردانند.

Attributes

صفت‌ها[1] در حقیقت اطلاعات توضیحی هستند که می‌توانید آنها را به برنامه‌های خود بیفزایید. صفتها را می‌توان برای کلیه عناصر برنامه از قبیل کلاسها، واسطها، اسمبلی ها و ... مورد استفاده قرار داد. از این اطلاعات می‌توان برای موارد متنوعی در زمان اجرای برنامه استفاده نمود.

برای مثال می‌توان به صفتی مانند DllImportAttribute اشاره کرد که امکان برقراری ارتباط با توابع کتابخانه‌ای Win۳۲ را فراهم می‌نماید.

همچنین صفتهایی نیز وجود دارند که برنامه‌نویس یا توسعه دهنده برنامه را در امر تولید برنامه یاری می‌نمایند. برای مثال می‌توان به صفت ObsoleteAttribute اشاره کرد که با استفاده از آن، در زمان کامپایل برنامه پیغامی برای برنامه نویس نمایش داده می‌شود و مشخص می‌کند که متدی خاص مورد استفاده قرار نگرفته و یا دیگر مورد استفاده نیست.

---

[1] Attributes

همچنین هنگامی‌که با فرمهای ویندوز کار می‌کنیم، صفتهای بسیاری وجود دارند که امکان استفاده از این فرمها را فراهم کرده و باعث می‌شوند تا اطلاعات مربوط به این عناصر در property فرم ظاهر شوند. یکی دیگر از موارد استفاده از صفتها در مسایل امنیتی اسمبلی‌های Net.[1] است.

برای مثال صفت‌هایی وجود دارند که باعث جلوگیری از فراخوانی‌های غیر مجاز می‌شوند، بدین معنی که تنها اجازه فراخوانی را به متدها یا اشیایی می‌دهند که قبلا تعریف شده و مشخص شده باشند.

یکی از علت‌های استفاده از صفت‌ها آن است که، اغلب سرویس‌هایی را که آنها برای کاربر فراهم می‌نمایند، بسیار پیچیده است و با کدهای معمولی نمی‌توان آنرا را بدست آورد. از اینرو استفاده از صفتها در بسیاری از موارد ضروری و اجتناب ناپذیر است.

---

[1] Assembly

همانطور که خواهید دید، صفتها به برنامه‌های ما متادیتا[1] اضافه می‌نمایند. پس از کامپایل برنامه‌های سی‌شارپ، فایل اسمبلی برای آن ایجاد می‌گردد که این اسمبلی معمولا یا یک فایل اجرایی است و یا یک Dll است. توصیف اسمبلی، در متادیتای مربوط به آن قرار می‌گیرد. طی پروسه‌ای تحت عنوان بازتاب[2]، صفت یک برنامه از طریق فایل متادیتای موجود در اسمبلی آن قابل دسترس می‌گردد.

در حقیقت صفت‌ها، کلاسهایی هستند که می‌توانید آنها را با زبان سی‌شارپ تولید کرده و جهت افزودن اطلاعاتی توضیحی به کد خود، از آنها استفاده نمایید. این اطلاعات در زمان اجرای برنامه از طریق بازتاب قابل دسترسی هستند.

---

[1] Metadata

[2] Reflection

مفاهیم اولیه درباره صفتها

صفتها را معمولا قبل از اعلان عنصر مورد نظر در برنامه قرار می‌دهند. اعلان صفتها بدین صورت است که نام صفت درون دو براکت ([]) قرار می‌گیرد.

```
[ObsoleteAttribute]
```

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

```
[Obsolete]
```

همچنین صفتها می‌توانند دارای پارامتر نیز باشند که با استفاده از آنها خواص بیشتری را در اختیار برنامه قرار می‌دهند. در ادامه موارد متنوعی از استفاده صفت ObsoleteAttribute را مشاهده می‌نمایید.

مثال:

```
using System;
class BasicAttributeDemo
{
    [Obsolete]
    public void MyFirstDeprecatedMethod()
    {
        Console.WriteLine("Called MyFirstDeprecatedMethod().");
    }
    [ObsoleteAttribute]
    public void MySecondDeprecatedMethod()
    {
        Console.WriteLine("Called MySecondDeprecatedMethod().");
    }
    [Obsolete("You shouldn't use this method anymore.")]
    public void MyThirdDeprecatedMethod()
    {
        Console.WriteLine("Called MyThirdDeprecatedMethod().");
    }
    // make the program thread safe for COM
    [STAThread]
```

```
    static void Main(string[] args)
    {
        BasicAttributeDemo attrDemo = new BasicAttributeDemo();
        attrDemo.MyFirstDeprecatedMethod();
        attrDemo.MySecondDeprecatedMethod();
        attrDemo.MyThirdDeprecatedMethod();
    }
}
```

همان‌طور که در مثال نیز مشاهده می‌شود، صفت Obsolete در فرمهای مختلف مورد استفاده قرار گرفته است. اولین محلی که از این صفت استفاده شده است، متد MyFirstDeprecatedMethod() و پس از آن در متد MySecondDeprecatedMethod() است. تنها تفاوت استفاده در این دو حالت آنست که در متد دوم صفت با نام کامل یعنی به همراه کلمه Attribute مورد استفاده قرار گرفته است. نتیجه هر دو اعلان یکسان است.

همانطور که گفته شد، صفتها می‌توانند دارای پارامتر نیز باشند :

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

همچنین این مثال شامل صفت دیگری نیز می‌باشد. این صفت STAThreadAttribute است که معمولا در ابتدای کلیه برنامه‌های سی‌شارپ و قبل از آغاز متد Main() قرار می‌گیرد. این صفت بیان می‌دارد که برنامه سی‌شارپ مورد نظر می‌تواند با کد مدیریت نشده COM از طریق Simple Threading Apartment ارتباط برقرار نماید.

استفاده از این صفت در هر برنامه‌ای می‌تواند مفید باشد، چراکه شما بعنوان برنامه نویس هیچ‌گاه اطلاع ندارید که آیا کتابخانه ثالثی که از آن استفاده می‌کنید، قصد برقراری ارتباط با COM را دارد یا نه؟ (در صورتیکه با برخی از اصطلاحات بکار رفته آشنایی ندارید اصلا نگران نشوید. در اینجا هدف تنها نشان دادن موارد استفاده از صفتهاست.)

صفتها می‌توانند دارای چندین پارامتر باشند. در این مثال استفاده از دو پارامتر برای یک صفت نشان داده شده است.

```
using System;
public class AnyClass
{
    [Obsolete("Don't use Old method, use New method", true)]
    static void Old() { }

    static void New() { }

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

همانطور که در مثال مشاهده می‌کنید، صفت مورد استفاده دارای دو پارامتر است. پارامتر اول که یک جمله متنی است و همانند مثال قبل عمل می‌کند. پارامتر دوم نیز بیان کننده نوع پیغامی است که این صفت در هنگام کامپایل تولید می‌کند. در صورتیکه این مقدار برابر با True باشد، بدین معناست که در هنگام کامپایل پیغام خطا تولید می‌شود و کامپایل برنامه متوقف می‌گردد. در حالت پیش فرض مقدار این پارامتر برابر با False است که بیان می‌دارد، به هنگام کامپایل تنها پیغام هشداری تولید خواهد شد. در پیغام این برنامه، عنصری از برنامه را که نباید از آن استفاده شود معین شده و جایگزین آن نیز معرفی می‌شود.

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

تفاوت پارامترهای positional با پارامترهای named در آن است که، پارامترهای named با نامشان مورد استفاده قرار می‌گیرند و همیشه اختیاری هستند.

صفت DllImport را مشاهده می‌نمایید که دارای هر دو نوع پارامتر positional و named است.

```
using System;
using System.Runtime.InteropServices;
class AttributeParamsDemo
{
    [DllImport("User32.dll", EntryPoint = "MessageBox")]
    static extern int MessageDialog(int hWnd, string msg, string caption,
intmsgType);

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

صفت DllImport در مثال فوق دارای یک پارامتر positional ("User32.dll") و یک پارامتر named (EntryPoint="MessageBox") است . پارامترهای named در هر مکانی می‌توانند قرار گیرند و مانند پارامترهای positional دارای محدودیت مکانی نیستند.

بدین معنا که چون در پارامترهای named، نام پارامتر مستقیما مورد استفاده قرار می‌گیرد، محل قرار گیری آن در لیست پارامترهای صفت مهم نیست اما در مورد پارامترهای positional چون اسم پارامتر مورد استفاده قرار نمی‌گیرد، این پارامترها حتما باید در مکانهای تعیین شده و تعریف شده در لیست پارامترهای صفت قرار گیرند. توجه کنید که چون هدف ما تنها آشنایی با صفتها و نحوه استفاده از آنهاست، درباره پارامترهای مختلف صفت DllImport بحث نخواهیم کرد چراکه پارامترهای این صفت نیاز به آشنایی کامل با API ۳۲Win دارد.

در یک بررسی کلی می‌توان گفت که پارامترهای Positional، پارامترهای سازنده[۱]صفت هستند و در هر بار استفاده از صفت باید مورد استفاده قرار گیرند، ولی پارامترهای Named کاملا اختیاری هستند و همیشه نیازی به استفاده از آنها نمی‌باشد.

---

[۱] Constructor

نشانگاه‌های صفتها (عناصری که صفتها بر روی آنها اعمال می‌شوند)

صفتهایی که تا کنون مشاهده کردید، همگی بر روی متدها اعمال شده بودند. اما عناصر مختلف دیگری در سی‌شارپ وجود دارند که می‌توان صفتها را بر روی آنها اعمال نمود.

جدول زیر عناصر مختلف زبان سی‌شارپ، که صفتها بر روی آنها اعمال می‌شوند را نشان می‌دهد.

عناصر اعمال شونده	قابل اعمال به
all	به تمامی عناصر قابل اعمال هستند.
assembly	به تمام یک اسمبلی
class	کلاسها
constructor	سازنده‌ها
delegates	Delegate ها
enum	عناصر شمارشی
event	رخدادها
field	فیلدها
interface	واسطها
method	متدها
module	ماژولها (کدهای کامپایل شده‌ای که می‌توانند به عنوان قسمتی از یک اسمبلی در نظر گرفته شوند.)
parameter	پارامترها
property	Property ها
returnvalue	مقادیر بازگشتی
struct	ساختارها

هر چند ممکن است استفاده از این نشانگاه‌ها[1]باعث ایجاد ابهام شوند، اما می‌توان با استفاده از آنها معین کرد که صفت دقیقا به عنصر مورد نظر اعمال شود.

یکی از صفتهایی که بر روی اسمبلی اعمال می‌شود و باعث ارتباط با CLS[2] می‌گردد، صفت CLSCompliantAttribute است. CLS امکان برقراری ارتباط بین کلیه زبانهایی که تحت .Net کار می‌کنند را فراهم می‌نماید. نشانگاه‌های صفتها با استفاده از اسم Target که بعد از آن کولون قرار می‌گیرد، ایجاد می‌شوند.

---

[1] Target

[2] Common Language Specification

در این مثال نحوه استفاده از این صفت نشان داده شده است.

```
using System;
[assembly: CLSCompliant(true)]
public class AttributeTargetDemo
{
    public void NonClsCompliantMethod(uint nclsParam)
    {
        Console.WriteLine("Called NonClsCompliantMethod().");
    }
    [STAThread]
    static void Main(string[] args)
    {
        uint myUint = 0;
        AttributeTargetDemo tgtDemo = new AttributeTargetDemo();
        tgtDemo.NonClsCompliantMethod(myUint);
    }
}
```

با استفاده از نشانگاه مورد نظر در اینجا یعنی assembly، این صفت بر روی کل اسمبلی اعمال می‌گردد. کد موجود در مثال فوق کامپایل نخواهد شد، زیرا uint در متد NonClsCompliantMethod() اعلان شده است.

در اینجا درصورتیکه فرم پارامتر صفت CLSCompliant را به false تغییر دهید و یا متد NonClsCompliantMethod() را به متدی منطبق با CLS تبدیل کنید (مثلا نوع بازگشتی آنرا int تعریف کنید) آنگاه برنامه کامپایل خواهد شد. (توضیحی که درباره CLS می‌توان بیان کرد این است که CLS مجموعه‌ای از ویژگیها و خواص .Net Framework است که به نحوی بیان می‌دارد، برای اینکه زبانهای مختلف تحت .Net بتوانند بدون مشکل با یکدیگر ارتباط برقرار نمایند، لازم است از یک سری از قوانین پیروی کنند، در غیر اینصورت امکان برقراری ارتباط با سایر کدهای نوسته شده تحت زبانهای برنامه‌سازی دیگر را نخواهند داشت.

برای مثال، استفاده از نوع uint به دلیل اینکه در زبانهای مختلف می‌تواند به صورتهای متفاوتی پیاده‌سازی شود و یا وجود نداشته باشد، سازگار با CLS نیست و برای اینکه بخواهیم برنامه‌ای منطبق با CLS داشته باشیم نباید از آن استفاده نماییم.)

نکته قابل توجه در مورد مثال مطرح شده آن است که در آن صفت CLSCompliant به استفاده از یک نشانگاه که همان assembly است، مورد استفاده قرار گرفته است و از این رو تمامی مشخصات این صفت به کلیه اعضای این اسمبلی اعمال خواهند شد. توجه نمایید که در این مثال علت و موارد استفاده از صفتها مشهودتر است، چراکه همانطور که مشاهده می‌نمایید، با استفاده از یک صفت می‌توانیم کنترلی بر روی کل اسمبلی و برنامه قرار دهیم تا در صورتیکه می‌خواهیم برنامه ما با سایر زبانهای برنامه‌سازی تحت .Net ارتباط برقرار کند، از متدهای استاندارد و سازگار با CLS استفاده نماییم که این قابلیت بزرگی را در اختیار ما قرار خواهد داد.

- صفت[1]ها کلاس هایی هستند که از کلاس Attribute ارث بری کرده‌اند.
- صفت‌ها استفاده می‌شوند تا بتوان رفتارهایی را به کلاس‌ها، متدها و حتی پراپرتی‌ها[2]ی برنامه اعمال نمود
- شیوه ی استفاده از صفت ها به شکل زیر می‌باشد

```
[Attribute_Name(argument(s))]
```

در ادامه به شرح نوع های مختلف صفت ها در سی شارپ می پردازیم

---

[1] Attribute

[2] Property

Obsolete

از این نوع صفت ها برای این منظور استفاده می‌شود که مشخص کنید آیا متد قابل استفاده (اجرا) می‌باشد.

Obsolete Attribute شامل دو ارگومان می‌شود.

i) message = پیام

ii) value = True/False

اگر مقدار value برابر با False باشد برنامه ی ما همراه با متد کامپایل شده و اجرا نیز خواهد شد

اگر مقدار value برابر با true باشد برنامه ی ما کامپایل شده ولی زمان اجرای متد برنامه ما مقدار Message را به صورت error نمایش می‌دهد

```
using System;
namespace Attribute
{
    class Program
    {
        [Obsolete("display1 is deprecated, use display2", false)]
        public void display1()
        {
            Console.WriteLine("Visual Studio 2008");
        }
        public void display2()
        {
            Console.WriteLine("Visual Studio 2015");
        }

        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.display1();
            Console.Read();
        }
    }
}
```

Conditional

از این نوع صفت‌ها در سی شارپ به این منظور استفاده می‌شود تا اعلام کنیم آیا متد ما باید اجرا شود یا خیر (این نوع صفت‌ها در سی شارپ برای جلوگیری از اجرای متد ها کاربرد دارد)

صفت های Conditional باید برروی متدهای یک کلاس یا استراکچر استفاده شوند و در صورت استفاده بر روی متدهای یک واسط[1] با خطا[2]زمان اجرا مواجه خواهیم شد.

---

[1] Interface

[2] error

متدهایی که اجرا شدن آن ها را به دست صفتهای Conditional وا می‌گذاریم باید خروجیشان از نوع void باشد.

```
#define hello
using System;
using System.Diagnostics;

namespace AttributeConditional
{
    class Program
    {
        [Conditional("hello")]
        public void print()
        {
            Console.WriteLine("Happy coding...");
        }
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.print();
            Console.Read();
        }
    }
}
```

نکته: صفت ها کاری روی بلاک کدهای شما انجام نمی‌دهند ولی برچسپی را به ان وصل می‌کنند که دیگر کدها می‌توانند آن را شناسایی کنند و براساس آن با کد یا کلاس شما رفتار کنند.

Abstract Classes

کلاسهای انتزاعی[1] کلاسهایی هستند که دارای یک یا چند متد پیاده سازی نشده هستند. اگر متدی را به شکل abstract تعریف کردید، باید کلاس را هم به شکل abstract تعریف کنید.

بطور مثال یک کلاس پایه با نام Shape تعریف می‌کنیم که پارامترهایی را نگهداری می‌کند و یک شی Pen هم برای رسم می سازیم. متد draw برای رسم استفاده می‌شود و آن را فقط تعریف می‌کنیم و کدی را برای پیاده سازی آن نمی نویسیم، چون هر شکلی به روش رسم خاصی نیاز دارد. (زمانی از کلاسهای Abstract استفاده می‌کنیم که بخواهیم کلاس پایه ای داشته باشیم و متدهای آن در کلاسهای مشتق شده نیاز به بازنویسی داشته باشه)

[1] Abstract Classes

به لیست کد ۱ دقت کنید:

```
// List Code 1
public abstract class Shape
{
    protected int height, width;
    protected int xpos, ypos;
    protected Pen bPen;
    public Shape(int x, int y, int h, int w)
    {
        width = w;
        height = h;
        xpos = x;
        ypos = y;
        bPen = new Pen(Color.Black);
    }
    public abstract void draw(Graphics g);
    public virtual float getArea()
    {
        return height * width;
    }
}
```

توجه کنید که متد draw را به شکل abstract تعریف کرده ایم و پس از تعریف با استفاده از ; انتهای تعریف متد را مشخص کرده ایم. کلاس تعریف شده نیز به شکل abstract تعریف شده است.

همچنین برای استفاده از کلاسهای انتزاعی نیز باید از کلاسهای مشتق شده آن یک نمونه بگیرید و از کلاس انتزاعی نمی‌توان یک نمونه ایجاد کرد.

ادامه مسیر ساخت کلاس را پی می‌گیریم و حال باید کلاس Rectangle را بسازیم. برای این منظور کلاس مورد نظر را از کلاس Shape که به شکل انتزاعی تعریف شده مشتق می‌کنیم. به لیست کد ۲ توجه فرمایید:

```
// List Code 2 (Rectangle class derived from abstract base class)
public class Rectangle : Shape
{
    public Rectangle(int x, int y, int h, int w) : base(x, y, h, w) { }
    //-----
    public override void draw(Graphics g)
    {
        g.DrawRectangle(bPen, xpos, ypos, width, height);
    }
}
```

کلاس Rectangle کلاسی است که می‌توانید از آن یک نمونه بگیرید. که یک متد draw واقعی دارد و می‌تواند رسم مستطیل را انجام دهد.

برای مثال کلاس دوم را هم پیاده سازی کرده ایم. به لیست کد ۳ دقت نمایید که پیاده سازی کلاس Circle را نمایش می‌دهد.

```
// List Code 3 (Circle class derived from abstract base class)
public class Circle : Shape
{
    public Circle(int x, int y, int r) : base(x, y, r, r) { }
    //-----
    public override void draw(Graphics g)
    {
        g.DrawEllipse(bPen, xpos, ypos, width, height);
    }
}
```

حال می‌خواهیم که بر اساس کلاس‌های تعریف شده، نمونه ای ایجاد کنیم. در لیست کد ۴ نمونه های تعریف شده اند که پارمترهای مورد نیاز با استفاده از سازنده آن ارسال شده است.

```
// List Code 4
public class Form1 : System.Windows.Forms.Form
{
    private PictureBox pictureBox1;
    private Container components = null;
    private Shape rect, circ;
    //-----
    public Form1()
    {
        InitializeComponent();
        init();
    }
    //-----
    private void init()
    {
        rect = new CsharpPats.Rectangle(50, 60, 70, 100);
        circ = new Circle(100, 60, 50);
    }
}
```

در پایان، در رویداد Paint رسم دایره و مستطیل را انجام می‌دهیم. به لیست کد ۴ دقت نمایید که یک دایره و یک مستطیل رسم شده است.

```
// List Code 5
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    rect.draw(g);
    circ.draw(g);
}
```

مقایسه کلاسهای انتزاعی و کلاسهای واسط

زمانی که یک کلاس واسط[1] تعریف می‌کنیم، مجموعه ای از تعریف یک یا چند متد را خواهیم داشت که باید برای هر کلاسی که از این واسط مشتق می‌شود کد نویسی شود. هیچ کد اولیه ای برای کلاسها نوشته نخواهد شد. در این حالت شما یک ساختار درختی از وراثت کلاسها خواهید داشت که از کلاس پایه (کلاس واسط) نمی‌توان نمونه ای ایجاد کرد و اگر متدی از واسط را در کلاس خارجی تعریف و کد نویسی نکنید حتما با خطای کامپایلر مواجه خواهید شد.

زمانی که یک کلاس انتزاعی[2] می سازید، یک کلاس پایه با تعریف یک یا چند متد دارید که ممکن است کد نویسی شده یا نشده باشند و به شکل abstract تعریف شده است. شما نمی‌توانید از یک کلاس انتزاعی نمونه بگیرید و البته می‌توانید از کلاس مشتق شده از کلاس انتزاعی که متدهای اون پیاده سازی شده است، نمونه بگیرید. اگر همه متدهای تعریف شده در کلاس بدون کد باشند، این کلاس کاملا مشابه کلاسهای واسط خواهد بود با این

---

[1] Interface Class

[2] Abstract Class

محدودیت که نمی‌توانید از آن مانند کلاس واسط ساختار درختی وراثت را داشته باشید. بیشترین استفاده از کلاسهای انتزاعی فراهم نمودن تعریف کلاس پایه ای است برای کلاسهای مشتق شده ای که می‌خواهند کاری را انجام دهند و به برنامه نویس اجازه داده می‌شود که شخصا کدهای موردنیازش را در کلاسهای مشتق شده مختلفی پیاده سازی نماید. استفاده دیگر پیاده سازی متدهای خالی از کد است که ممکن است در کلاسهای مشتق شده دیگر پیاده سازی نشده باشند و بخواهید اطمینان داشته باشید که پروژه شما کامپایل خواهد شد.

به مثال زیر توجه نمایید:

```
public class NullShape
{
    protected int height, width;
    protected int xpos, ypos;
    protected Pen bPen;
    //-----
    public Shape(int x, int y, int h, int w)
    {
        width = w;
        height = h;
        xpos = x;
        ypos = y;
        bPen = new Pen(Color.Black);
    }
    //-----
    public void draw(Graphics g) { }
    //-----
    public virtual float getArea()
    {
        return height * width;
    }
}
```

در این مثال متد draw خالی رها شده و کدی برای آن نوشته نشده است و کلاس مشتق شده بدون مشکل کامپایل خواهد شد. و هیچ اشاره ای به اینکه این متد یک متد انتزاعی بازنویسی شده است نمی‌شود.