# **Enhancement in Out Variables in C# 7**

در نسخه های قدیمی تر از سی شارپ 7 زمانی که از کلمه کلیدی out استفاده می کردیم لازم بود که ابتدا متغییر های مورد نیاز را تعریف کنیم اما لازم نبود که آن ها را init کنیم اما و سپس پاس می دادیم به متد اما در نسخه ی 7 به بعد لازم نیست که یک مرحله متغییر را تعریف کنیم و سپس آن را به متد با کلمه out پاس بدهیم بلکه کافی است در هنگام پاس دادن تعریف کنیم.

GetEmployeeDetails**(**out **string** EmployeeName, out **string** Gender, out **long** Salary**)**;

Scope این ها در محدود brace هستند. {}

همچنین امکان تعریف با استفاده از کلمه کلیدی var نیز مسیر است بدون نیاز به تعریف نوع داده به صورت صریح.

**How to Ignore an Out parameter in C#?**

با استفاده از underscore این امکان وجود دارد که یک پارامتر را ignore کرد.

GetEmployeeDetails**(**out var EmployeeName, out var Gender, out \_**)**;

**Try Pattern using Out Variable**

یکی از ویژگی های خوب اضافه شده try pattern است که برای تبدیل نوع ها کاربرد دارد که اگر عملیافت موفق باشد به مقدار دسترسی خواهیم داشت.

**if** **(**DateTime.TryParse**(**s, out DateTime date**))**

**{**

Console.WriteLine**(**date**)**;

**}**

# **Pattern Matching in C#**

مکانیسمی است برای تشخیص این که یک valueایا از شکل خاصی است یا نه. آیا یک Value دارای type خاصی است یا نه؟

کلمات کلیدی

1. Pattern Matching using “**is**” expression
2. The Pattern Matching using “**case**” statements

قابلیت استفاده در انواع data type را دارد حتی custom ها و همچنین قابلیت extract داده از expression را نیز دارد.

نکته : if/else فقط قابلیت کار با primitive ها را دارند.

##### **Pattern Matching in C# with “is” expression**:

برای بررسی سازگاری یک نوع کاربرد دارد مثلا یک interface را بررسی کنیم که implement شده است یا نه. یک نوع از نوع دگیری ارث برده است یا نه....

مثال:

یک کلاس بیس به نام shape ساخته و 3 کلاس از آن ارث برده اند حالا با استفاده از is می خواهیم تست کنیم از کدام نوع است کلاس ورودی ما؟

**static** **void** Main**()**

**{**

Circle circle = new Circle**(**10**)**;

DisplayArea**(**circle**)**;

Rectangle rectangle = new Rectangle**(**10, 5**)**;

DisplayArea**(**rectangle**)**;

Triangle triangle = new Triangle**(**10, 5**)**;

DisplayArea**(**triangle**)**;

Console.ReadKey**()**;

**}**

**public** **static** **void** DisplayArea**(**Shape shape**)**

**{**

//Checking the actual type of the shape object

**if** **(**shape **is** Circle**)**

**{**

//then typecast to actual Circle type

Circle c = **(**Circle**)**shape;

Console.WriteLine**(**"Area of Circle is : " + c.Radius \* c.Radius \* Shape.PI**)**;

**}**

**else** **if** **(**shape **is** Rectangle**)**

**{**

//then typecast to actual Rectangle type

Rectangle r = **(**Rectangle**)**shape;

Console.WriteLine**(**"Area of Rectangle is : " + r.Length \* r.Height**)**;

**}**

**else** **if** **(**shape **is** Triangle**)**

**{**

//then typecast to actual Triangle type

Triangle t = **(**Triangle**)**shape;

Console.WriteLine**(**"Area of Triangle is : " + 0.5 \* t.Base \* t.Height**)**;

**}**

**else**

**{**

**throw** new ArgumentException**(**message: "Invalid Shape", paramName: **nameof(**shape**))**;

**}**

**}**

روش دوم و بهتر patternMatching بدون نیاز به case :

دسترسی مستقیم به object در صورت برقراری شرط است.

**if** **(**shape **is** Circle c**)**

**{**

Console.WriteLine**(**"Area of Circle is : " + c.Radius \* c.Radius \* Shape.PI**)**;

**}**

مثال قبلی با استفاده از کلمه کلیدی Switch با قابلیت دریافت object بدون cast

**switch** **(**shape**)**

**{**

**case** Circle c:

Console.WriteLine**(**"Area of Circle is : " + c.Radius \* c.Radius \* Shape.PI**)**;

**break**;

**case** Rectangle r:

Console.WriteLine**(**"Area of Rectangle is : " + r.Length \* r.Height**)**;

**break**;

**case** Triangle t:

Console.WriteLine**(**"Area of Triangle is : " + 0.5 \* t.Base \* t.Height**)**;

**break**;

**default**:

**throw** new ArgumentException**(**message: "Invalid Shape", paramName: **nameof(**shape**))**;

**case** **null**:

**throw** new ArgumentNullException**(nameof(**shape**))**;

**}**

نکته : در استفاده از Case همیشه دقت کنید default را در آخر بگذارید.

##### **Case Expressions using When Clauses in C#:**

این کلمه کلیدی در switch case کاربرد دارد برای زمانی که بخواهیم یک شرط نیز به case خود اضافه کنیم برای بررسی :

**switch** **(**shape**)**

**{**

**case** Rectangle r **when** r.Length == r.Height:

Console.WriteLine**(**"Area of Sqaure is : " + r.Length \* r.Height**)**;

**break**;

**case** Rectangle r:

Console.WriteLine**(**"Area of Rectangle is : " + r.Length \* r.Height**)**;

**break**;

**case** Circle c:

Console.WriteLine**(**"Area of Circle is : " + c.Radius \* c.Radius \* Shape.PI**)**;

**break**;

**case** Triangle t:

Console.WriteLine**(**"Area of Triangle is : " + 0.5 \* t.Base \* t.Height**)**;

**break**;

**default**:

**throw** new ArgumentException**(**message: "Invalid Shape", paramName: **nameof(**shape**))**;

**}**

**}**

# **Digit Separators in C# 7**

در دنیای واقعی خواندن اعداد بزرگ کار سختی است در دنیای برنامه نویسی نیز برای حل این مشکل از digit separator استفاده می شود که با underscore قابلیت جدا کردن اعداد را از هم می دهد برای خوانایی بهتر .

نکته : امکان استفاده با تمامی نوع های عددی را دارد. همچنین الزام به استفاده فقط از یک underscore نیست بلکه می توان چندین underscore اضافه کرد.

var bigNumberSplit = 123\_456\_789\_012\_345\_678;

var myData1 = 1\_\_2\_\_\_\_\_\_\_\_3\_\_\_4\_\_\_\_5\_\_\_\_\_6;

var myData2 = 1\_\_2\_\_\_3\_\_\_4\_\_\_\_5\_\_\_\_\_6.79;

# **Tuples in C# 7**

زمانی که می خواهید بیش از یک مقدار از یک متد برگردانید یکی از راهکار های که می توان از آن استفاده کرد tuple است.

مثال

//Declaring the return type as Tuple<int, double>

**private** **static** Tuple**<int**, **double>** Calulate**(**IEnumerable**<double>** values**)**

**{**

**int** count = 0;

**double** sum = 0.0;

**foreach** **(**var **value** in values**)**

**{**

count++;

sum += **value**;

**}**

//Creating an object of Tuple class by calling the static Create method

Tuple**<int**, **double>** t = Tuple.Create**(**count, sum**)**;

//Returning the tuple instance

**return** t;

**}**

نکته : tuple ها به صورت referenceType هستند و حافظه heap به آن ها تخصیص داده می شود و زمانی gb حافظه را پاک می کند که به آن داده نیازی نباشد برای مواقعی performace لازم است بهتر است استفاده نشود.

نکته : به دلیل این که عناصر موجود در tuple اسم نمی گیرند ممکن است سبب ناخوانایی کد شود. اسم آن ها صرفا item1 item2 خواهد بود.

**How To Solve**

استفاده از tupe به همراه نام

**private** **static** **(int** count, **double** sum**)** Calulate**(**IEnumerable**<double>** values**)**

**{**

**int** count = 0;

**double** sum = 0.0;

**foreach** **(**var **value** in values**)**

**{**

count++;

sum += **value**;

**}**

**return** **(**count, sum**)**;

**}**

# **Splitting Tuples in C# 7**

می توان مقادیر tuple را به صورت کامل در var ریخته و یکی یکی جدا کنیم.

//GetEmployeeDetails Method returning a Tuple

var EmployeeDetails = GetEmployeeDetails**(**1001**)**;

//Storing Each Value of Tuple in Separate Variables

var Name = EmployeeDetails.Item1;

var Salary = EmployeeDetails.Item2;

var Gender = EmployeeDetails.Item3;

var Dept = EmployeeDetails.Item4;

**private** **static** **(string**, **double**, **string**, **string)** GetEmployeeDetails**(long** EmployeeID**)**

**{**

//Based on the EmployeeID get the data from a database

//Here we are hardcoded the value

**string** EmployeeName = "Pranaya";

**double** Salary = 2000;

**string** Gender = "Male";

**string** Department = "IT";

**return** **(**EmployeeName, Salary, Gender, Department**)**;

**}**

می توانیم برای هر مقداری که از متد برمیگردد یک name , type مشخص کنیم :

//De-Constructing Tuples

**(string** Name, **double** Salary, **string** Gender, **string** Dept**)** = GetEmployeeDetails**(**1001**)**;

روش 2 می توانیم با ترکیب کلمه کلیدی Var

var **(**Name, Salary, Gender, Dept**)** = GetEmployeeDetails**(**1001**)**;

روش سوم استفاده از var به صورت تک به تک

**(**var Name, var Salary, **string** Gender, var Dept**)** = GetEmployeeDetails**(**1001**)**;

روش چهارم استفاده از ( )

**(**Name, Salary, Gender, Dept**)** = GetEmployeeDetails**(**1001**)**;

# **Ref Returns and Ref Locals in C# 7**

**از کلمه کلیدی ref در گذشته به عنوان ورودی متد استفاده می کردیم اما در نسخه ی 7 به بعد سی شارپ می توان به عنوان return type نیز استفاده کرد.**

#### **Passing by Value in C#**

اگر پارامتر value type داشته باشیم value ما در scope مربوط به متد ایجاد می شود و در انتهای اجرای متد destroy می شود و تاثیری در مقدار ارسالی ندارد چون کپی آن ارسال شده است.

#### **Passing by Reference in C#**

اگر بخواهیم پارامتر ارسالی تغییر کند باید از کلمه کلیدی ref استفاده کنیم تا تغییرات انجام شده بر روی داده ی اصلی نیز تاثیر گذار باشد.

**int** Num = 1;

//Calling the PassByReference Method using Pass By Reference Mechanism

PassByReference**(ref** Num**)**;

**static** **void** PassByReference**(ref** **int** Number**)**

**{**

//Changing Number will change the memory address value which is created by Num

Number = 2;

**}**

#### **Out Variable in C#:**

در سناریو های که فقط لازم باشد یک مقدار از متد ما برگردد ولی ما مقداری ندهیم باید از out استفاده کرد. به دلیل این که آرگومان و پارامتر از یک رفرنس استفاده می کنند پس تغییرات روی داده اصلی تاثیر می گذارد.

**int** Num = 1;

//Calling the OUT Method using Pass By Reference Mechanism

OUT**(**out Num**)**;

**static** **void** OUT**(**out **int** Number**)**

**{**

//Changing Number Variable will chnage the Num Variable

Number = 2;

**}**

می توان بدون init هم صدا زد

OUT**(**out **int** Num**)**;

#### **Ref Local in C# 7**

کلمه کلیدی ref یک type جدید است برای ذخیره کردن reference برای ذخیره کردن آدرس کاربرد دارد

**int** Number1 = 1;

//Create a Local variable using ref keyword and store the referenec of another variable

**ref** **int** Number2 = **ref** Number1;

Number2 = 2;

اتقاقی که در کد بالا رخ داده است سبب این می شود که با تغییر number2 مقدار number1 نیز تغییر کند. چون گفته ایم تو reference خوردی به مقدار number1

#### **Ref Returns in C# 7**

به عنوان توسعه دهنده نیاز داریم که بتوانیم از یک متد reference را به عنوان return type برگردانیم با استفاده از کلمه کلیدی ref این موضوع امکان پذیر است.

با استفاده از کد زیر مقدار درون آرایه را با استفاده از دسترسی به reference آن تغییر دادیم بدون این که به خود آرایه مستقیما دست بزنیم.

**public** **ref** **int** GetFirstOddNumber**(int[]** numbers**)**

**{**

**for** **(int** i = 0; i **<** numbers.Length; i++**)**

**{**

**if** **(**numbers**[**i**]** % 2 == 1**)**

**{**

//Returning the Reference of the Number

**return** **ref** numbers**[**i**]**;

**}**

**}**

**throw** new Exception**(**"Odd Number Not Found"**)**;

**}**

**ref** **int** OddNumber = **ref** program.GetFirstOddNumber**(**IntegerArray**)**;

//Changing the oddNum value as it is a reference variable,

//it will also change the same in the actual Integer Array

OddNumber = 35;

# **Local Functions in C# 7**

فانکشن های هستند که به صورت داخلی در یک فانکشن پدر تعریف می شوند و به صورت private هستند و فقط قابلیت فراخوانی توسط فانکشن پدر خود را دارند.

در مواقعی که به یک helper داخل یک متد نیاز داریم که کار کوچکی را انجام دهد یا validation بخواهیم و همچنین در revursive ها نیز به صورت داخلی تعریف شوند فضای حافضه ی کمتری مصرف خواهد کرد.

1. You can not overload a Local Function in C#
2. The Accessibility modifiers such as public, private, and protected are not allowed.
3. The compiler will issue a warning if the local function is not used by the parent function as there is no meaning of defining a local function in C# if it is not being used by the parent method.
4. All variables in the enclosing scope, including local variables, can be accessed

# **Generalized Async Return Types in C# 7**

Why Task instead of void?

The difference between them is that if you use the Void Return Type then the async method cannot be awaited. That means the caller of such a method (void return async method) does not have any option to wait for the async method to complete its work. They simply call the async method and continue their work. So, if you have methods other than event handlers that don’t return any value, it’s always advisable to use Task return type instead of void. On the other hand, if you use Task as the return type of an async method, then you can await the method call i.e. the caller of the method can wait till the async method completes its execution.

برای جلوگیری از تخصیص حافظه اضافی در Task به دلیل این که از نوع referece است می توان از یک نسخه ی lightweight استفاده کرد به نام VlaueTask<T> . performance بهتری دارد به دلیل ValueType بودن.

# **Expression Bodied Members in C#**

برای تمامی method ها یا property های که logic آن ها در یک expression می تواند قرار گیرد کاربرد دارد.

**member => expression;**Where expression is a valid C# expression.

1. Methods
2. Properties
3. Constructor
4. Destructor
5. Getters
6. Setters
7. Indexers

مثال

**public** **string** GetFullName**()** =**>** $"{FirstName} {LastName}";

مثال Ctor

**public** Location**(string** name**)** =**>** LocationName = name;

مثال کامل

**public** **class** Location

**{**

//Private Variable to store the Location Name

**private** **string** \_locationName;

//Constructor Expression Bodied Member to set the private variable \_locationName

**public** Location**(string** name**)** =**>** \_locationName = name;

//You can also use the following to set the public property LocationName

//public Location(string name) => LocationName = name;

//Expression Bodied Member for Read-Only Property

**public** **string** LocationName =**>** \_locationName;

**}**

**Thrown Expression Sample**

**public** Employee**(string** name**)** =**>**

FullName = name ?? **throw** new ArgumentNullException**(**name**)**;

## **Async Main in C# with Examples**

From C# 7.1, the Main() method, which is the application’s entry point, can be declared async. Before C# 7.1, the Main() method can have a return type as either void or int; however, now, it also supports **Task** and **Task<int>**. So, From C# 7.1, we now have eight overload versions that are considered valid signatures for the **Main()** method, as shown below.

**public** **static** **void** Main**()**;

**public** **static** **int** Main**()**;

**public** **static** **void** Main**(string[]** args**)**;

**public** **static** **int** Main**(string[]** args**)**;

**public** **static** Task Main**()**;

**public** **static** Task**<int>** Main**()**;

**public** **static** Task Main**(string[]** args**)**;

**public** **static** Task**<int>** Main**(string[]** args**)**;

**static** **async** Task Main**(string[]** args**)**

**{**

Console.WriteLine**(**"From C# 7.1, Async Main Method"**)**;

Console.WriteLine**(**$"Main Method execution started at {System.DateTime.Now}"**)**;

//Calling Async Method using await as now the Main method is async

**await** SomeAsyncMethod**()**;

Console.WriteLine**(**$"Main Method execution ended at {System.DateTime.Now}"**)**;

Console.ReadKey**()**;

**}**

# **ReadOnly Structs in C#**

از کلمه کلیدی readOnly برای مواقعی استفاده می کنیم که بخواهیم یک filed فقط در constractor مقدار بگیرد و از آن به بعد immutable شود.

**Read-Only Structs in C#**

اگر در تعریف یک strauct از کلمه کلیدی readOnly استفاده شده باشد.

1. The members of the struct are read-only.
2. None of the members can have setters i.e. they only have getters.
3. A parameterized constructor must be used to initialize the data members of a struct.
4. The struct is immutable.
5. The “this” variable cannot be changed in any other method except the constructor. That means the struct readonly members can only be changed through the constructor.

اگر بخواهیم تمام strauct تبدیل به readOnly نشود می توانیم آن را ققط به اعضا بدهیم

نکته : یک فیلد می تواند با expression body مقدار بگیرد =>

**public** **double** Height **{** **get**; **set**; **}**

**public** **double** Width **{** **get**; **set**; **}**

//Property Expression Bodied Member

**public** **double** Area =**>** **(**Height \* Width**)**;

**نکته :** اگر property داشته باشیم که فقط قابلیت Get دارد و نمی توان set از بیرون برای آن انجام داد باید به صورت readOnly تعریف شود به دلیل این که compiler تشخیص می دهد و warning می دهد.

اگر یک struct را به صورت readOnly تعریف کنیم لازم است که کل اعضای آن نیز readOnly باشند.

**public** **readonly** struct Rectangle

**{**

**public** **readonly** **double** Height **{** **get**; **}**

**public** **readonly** **double** Width **{** **get**; **}**

# **Default Interface Methods in C#**

قابلیتی است که به کمک آن می توان برای اعضای interface مقدار DefaultMehtod قرار داد این ویژگی از سی شارپ 8 به بعد افزوده شده و قبلا در انحصار abstract class ها بود.

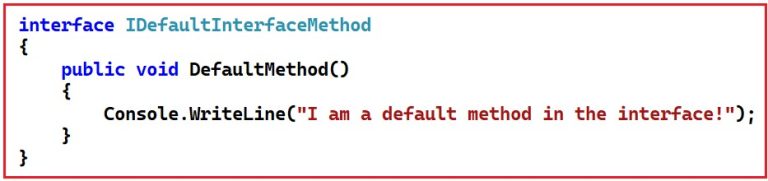
این قابلیت به ما این امکان را می دهیم که توسعه ی ویژگی های interface را انجام بدهیم بدون این که عدم سازگاری ایجاد شود برای کسانی که از نسخه های قبلی استفاده می کنند.

**Allowed in the Interface in C#:**

1. A body for a method or indexer, property, or an event accessor
2. Private, protected, internal, public, virtual, abstract, sealed, static, extern
3. Static fields
4. Static methods, properties, indexers, and events.
5. Explicit access modifiers with default access are public

**Not allowed in the interface in C#:**

1. Instance state, instance fields, instance auto-properties
2. override keyword is currently not possible, but this might be changed in C# 9



**زمانی که در یک interface یک DefaultMehtod بنویسیم دیگر در implementation خطای مبنی بر این که پیاده سازی نشده داریم نمی دهد. همچنین امکان فراخوانی آن وجود دارد اما خطای Compile می دهد چون کلاس inherited هیچ ایده از وجود پیاده سازی پیش فرض ندارد اما در Abstarct امکان دسترسی وجود دارد.**

**امکان تعریف متد Virtual و override کردن آن در interface وجود دارد.**

**interface IDefaultInterfaceMethod**

**{**

**// By default, this method is virtual. The virtual keyword is not required here**

**virtual void DefaultMethod()**

**{**

**Console.WriteLine("I am a default method in the interface!");**

**}**

**// By default, this method is abstract and public, so the abstract and public keyword not required here**

**public abstract void Sum();**

**}**

**نکته : یک رابط می تواند از یک رابط دیگر ارث بری کند.**

**Pattern Matching in C#**

برای بررسی یک type کاربر دارد. در نسخه ی 7 با استفاده از Expression و swtich امکان پذیر بود.

ویژگی های جدید مربوط به سی شارپ 8 برای pattern matching

1. **Switch Expressions**
2. **Property Patterns**
3. **Tuple Patterns**
4. **Positional Patterns**

مزیت اصلی این است که به جای استفاده مداوم از if else های تو در تو می توانیم به صورت ساده پیاده سازی کنیم.

**Switch Expressions in C#**

var message = day.ToUpper**()** **switch**

**{**

"SUNDAY" =**>** "Weekend",

"MONDAY" =**>** "start of the weekday",

"FRIDAY" =**>** "end of the weekday",

\_=**>** "Invalid day"

**}**;

مقدار را از day.ToUpper می خواند و برگشتی را در message می ریزد.

**Property Patterns in C# 8**

برای بررسی و مقایسه مقادیر property کاربرد دارد. در مثال زیر مقادیر بررسی می شوند که اگر نوع خاص با مقدار خاص باشد شرط برقرار است.

**Public** **static** **void** IsSpecificRectanglee**(**Rectangle rectangle**)**

**{**

**if** **(**rectangle **is** Rectangle **{** Length: 5, Height: 10 **}** specificRectangle**)**

**{**

Console.WriteLine**(**$”Rectangle Length: {specificRectangle.Length} and Height: {specificRectangle.Height}”**)**;

**}**

**}**

می توان به روش ساده تر زیر هم نوشت برای بررسی مقادیر

**if** **(**rectangle **is** Rectangle **{**Length: 5, Height: 10**})**

**Tuple Patterns in C# 8**

برای بررسی مچ بودن چندین مقدار و منطق های پیچیده به روشی ساده کاربر دارد. برای بررسی یک ترکیب که به صورت tuple آمده است با استفاده از switch کاربرد دارد.

Var result = **(**sport1, sport2, sport3**)** **switch**

**{**

//matches if Cricket, Football, and Swimming given as input

**(**“Cricket”, “Football”, “Swimming”**)** =**>** “I like Cricket, Football, and Swimming.”,

//matches if Cricket, Football, and Baseball given as input

**(**“Cricket”, “Football”, “Baseball”**)** =**>** “I like Cricket, Football, and Baseball.”,

//matches if Hockey, Football, and Swimming given as input

**(**“Hockey”, “Football”, “Swimming”**)** =**>** “I like Hockey, Football, and Swimming.”,

//matches if Table Tennis, Football, and Swimming given as input

**(**“Table Tennis”, “Football”, “Swimming”**)** =**>** “I like Table Tennis, Football and Swimming.”,

//Default case

**(**\_, \_, \_**)** =**>** “Invalid input!”

**}**;

مثال 2

**public** **static** **int** GetOrderDiscount**(**CustomerOrder order**)**

**{**

**return** **(**order.PaymentMethod, order.Country**)** **switch**

**{**

**(**PaymentMethods.CreditCard, "India"**)** =**>** 20,

**(**PaymentMethods.WireTransfer, "USA"**)** =**>** 15,

**(**\_, \_**)** **when** order.Amount **>** 5000 =**>** 10,

\_ =**>** 0 // unknown or Default

**}**;

**}**

**Positional Patterns in C# 8**

**Deconstruct**

**Rectangle rectangle = new Rectangle { Length = 20, Height = 40 };**

**var (length, height) = rectangle;**

**Console.WriteLine($"The rectangle Length: {length} and Height: {height}");**

**if (rectangle is (20, \_) rect)**

**{**

**Console.WriteLine("The rectangle has a length of 20");**

**}**

**public class Rectangle**

**{**

**public double Length { get; set; }**

**public double Height { get; set; }**

**public void Deconstruct(out double length, out double height)**

**{**

**length = Length;**

**height = Height;**

**}**

**}**

**------------------------------------------------------------------**

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

**{**

**Customer customer = new Customer()**

**{**

**FirstName = "Sam",**

**LastName = "Taylor",**

**Email = "info@dotnettutorials.net",**

**CustomerAddress = new Address() { PostalCode = 755019, Street = "Newyork", Country = "USA"}**

**};**

**Console.WriteLine($"Is Free Shipping Eligible : {IsFreeShippingEligible(customer)}");**

**}**

**public static bool IsFreeShippingEligible(Customer customer)**

**{**

**// If customer is from USA then Free shipping applies.**

**return customer is Customer(\_, \_, \_, (\_, \_, "USA"));**

**}**

**}**

**public class Customer**

**{**

**public string FirstName { get; set; }**

**public string LastName { get; set; }**

**public string Email { get; set; }**

**public Address CustomerAddress { get; set; }**

**//Deconstruct for Customer**

**public void Deconstruct(out string firstname, out string lastname, out string email, out Address address)**

**{**

**firstname = FirstName;**

**lastname = LastName;**

**email = Email;**

**address = CustomerAddress;**

**Using Declarations in C#**

از کلمه کلیدی using برای تعریف متغییر های از نوع disposable استفاده می کنیم یعنی ایجاد instance از شی که IDisposable را پیاده سازی کرده است. مثل io , web service از این روش برای اطمینان از dispose شدن شی استفاده می کنیم.

تنها مشکل افرودن using به کد ایجاد Scope جدید برای کد بوده است که در سی شارپ 8 این مشکل رفع شده است و همچنین تضمین شده است که حتی در صورت بروز exception نیز فراخوانی متد dispose انجام شود.

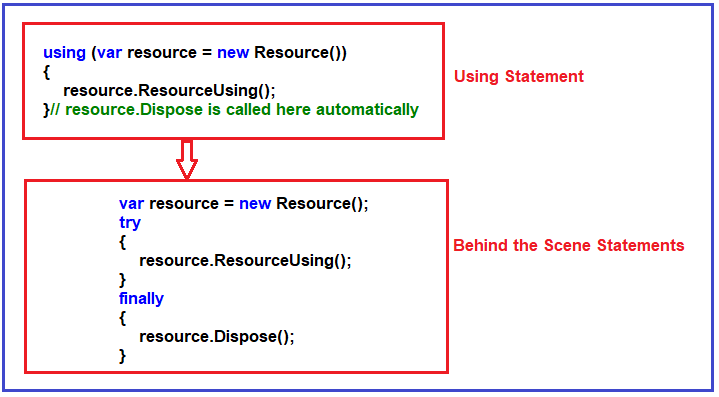
با استفاده از این کلمه به compiler می گوییم که متغییر ما باید در انتهای اجرای کد dispose شود یعنی انتهای scope خود.

Idisposable

یک مکانیسم برای رها کردن حافظه های unmanaged است.

**How is the Dispose Method Automatically Called in C#?**

زمانی که از using استفاده می کنیم پشت صحنه کامپایلر یک بلوک کد try/finally ایجاد می کند برای اطمینان از فراخوانی حتی در زمانی که exception رخ می دهد finally تضمین می دهد که بروز Ex داشته باشیم چه نه عملیات انجام شود.



در نسخه ی جدید نیاز به تعریف {} نیست به دلیل این که در انتهای Scope به صورت خودکار فراخوانی dispose با استفاده از کلمه کلیدی using و ایجاد try/finally انجام می شود.

//Creating an Instance of Resource Class with the new using declaration

**using** var resource = new Resource**()**;

سینتکس قدیم برای حالت چندتایی در انتهای {} متغییر dispose می شود.

**using** **(**var resource1 = new Resource**())**

**{**

**using** **(**var resource2 = new Resource**())**

**{**

resource1.ResourceUsing**()**;

resource2.ResourceUsing**()**;

**}**//Here, the resource2.Dispose() Method is called automatically

**}**//Here, the resource1.Dispose() Method is called automatically

Console.WriteLine**(**"Main Method End..."**)**;

**}**

New syntax

**using** var resource1 = new Resource**()**;

**using** var resource2 = new Resource**()**;

**How to Dispose of a Resource Before the Method Ends in C# by Using Declarations?**

در این حالت لازم است که {} را استفاده کنیم تا در انتها dispose انجام شود.

**using** var resource1 = new Resource**()**;

resource1.ResourceUsing**()**;

//Creating a Block using only curly braces

**{**

**using** var resource2 = new Resource**()**;

resource2.ResourceUsing**()**;

**}**//Here, resource2.Dispose() Method is called automatically

**Static Local Functions in C#**

**public** **static** **void** Calculate**()**

**{**

**int** X = 20, Y = 30;

CalculateSum**(**X, Y**)**;

// Here CalculateSum is the local function of the Calculate function

**static** **void** CalculateSum**(int** Num1, **int** Num2**)**

**{**

**int** sum = Num1 + Num2;

Console.WriteLine**(**$"Num1 = {Num1}, Num2 = {Num2}, Result = {sum}"**)**;

**}**

// Calling Static Local function

CalculateSum**(**30, 10**)**;

CalculateSum**(**80, 60**)**;

**}**

**Disposable Ref Structs in C#**

از نسخه ی 7 سی شارپ امکان تعریف struct از نوع ref فراهم شده است و این امکان را می دهد که حافظه stack اختصاص داده شود و از اختصاص heap جلوگیری شود.

امکان ارث بری از interface توسط struct وجود ندارد.

اکنون امکان تعریف disposable برای struct وجود دارد بدون پیاده سازی Idisposable

**IDisposable Interface in C#:**

برای آزاد سازی unmanaged resource کاربرد دارد.

//Ref struct Cannot Implement the IDisposable Interface

//hence cannot provide Implementation for Dispose method

**ref** struct Rectangle : IDisposable

**{**

**public** **void** Dispose**()**

**{**

**}**

**}**

کد بالا خاط می دهد به دلیل پیاده سازی رابط درست کد به صورت زیر است.

**//Ref struct Cannot Implement the IDisposable Interface**

**//hence cannot provide Implementation for Dispose method**

**ref struct Rectangle**

**{**

**public void Dispose() { }**

**}**

از نسخه ی 8 به بعد امکان تعریف Dispose بدون پیاده سازی رابط Idisposable وجود دارد.

**Nullable Reference Types in C# 8**

تا قبل از نسخه ی 8 تمامی reference type ها به صورت nullable بودند اما از نسخه ی 8 به بعد سی شارپ امکان این که مشخص کنیم nullable باشند یا نه فراهم شده.

زمانی که یک instance از یک reference type ایجاد می کنید یک object در heap ایجاد می شود و اگر به هیج مقدار point نکند مقدار null میگیرد به دلیل null reference type

در نسخه ی 8 به بعد امکان مشخص کردن نال پذیری وجود دارد برای Reference type ها.

**How to Enable Nullable Reference Types in C#?**

برای این که از هشدار استفاده از متغییر های نال پذیر استفاده کنیم باید آن را فعال کنیم.

**.csproj** file and add **<Nullable>enable</Nullable>** in the property group as shown in the below image.

با فعال کردن این ویژگی احتمال و با دریافت هشدار مقادیر null احتمال این که NullReferenceException مواجه شوید کم می شود.

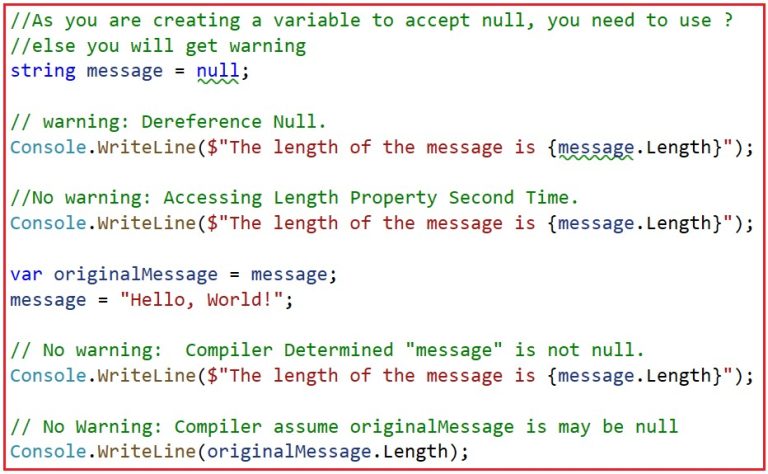
**راه های تشخیص این که کامپایلر تشخیص دهد مقداری نال است یا نه؟**

**به متغییر یک مقدار assign شده که نال نیست.**

**متغییر در مقابل این که نال است یا نه بررسی شده و قبل چک استفاده نشده است**

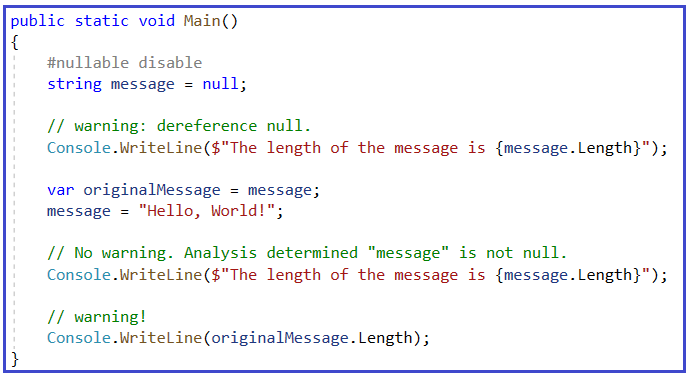
**هر متغییری که تشخیص ندهد دارای مقدار null است کامپایلر به عنوان my be null تشخیص می دهد.**

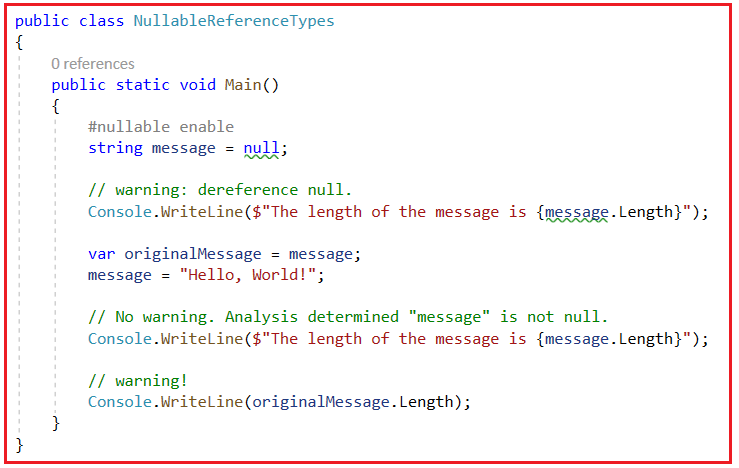
زمانی که به یک متغییر که احتمال null بودن دارد به صورت تصادفی دسترسی پیدا می کنیم کامپایلر اخطار می دهد که احتمال null بودن است. برای اولین بار هشدار می دهد و دفعات بعدی نه.



برای جلوگیری از هشدار مقدار null می توان از دارکتیو استفاده کرد

#nullable disable





**Rules for Non-nullable Reference Type in C#**

When a variable is not supposed to be null, the compiler enforces some rules to make sure that it is safe to dereference that variable without checking that it is not null.

1. The variable must be initialized to a non-null value.
2. The variable can never be assigned the null value.

**Rules for Nullable Reference Type in C#**

When a variable can be null, in that case, the compiler enforces different rules to make sure that you have correctly checked for a null reference.

1. The variable may only be dereferenced when the compiler can guarantee that the value is not null.
2. It may be initialized with the default null value and may be assigned the value null in another code.

**Asynchronous Streams in C#**

ویژگی اضافه شده به یس شارپ نسخه 8 است که راه حلی برای پشتیبانی Async از streams و IEnumerable data را فراهم می کند به طور مثال IAsyncEnumerable استفاده می شود.

از سی شارپ 8 امکان استفاده از stream ها به صورت Async فراهم شد.

متدی که async stream برمیگرداند دارای ویژگی های زیر است.

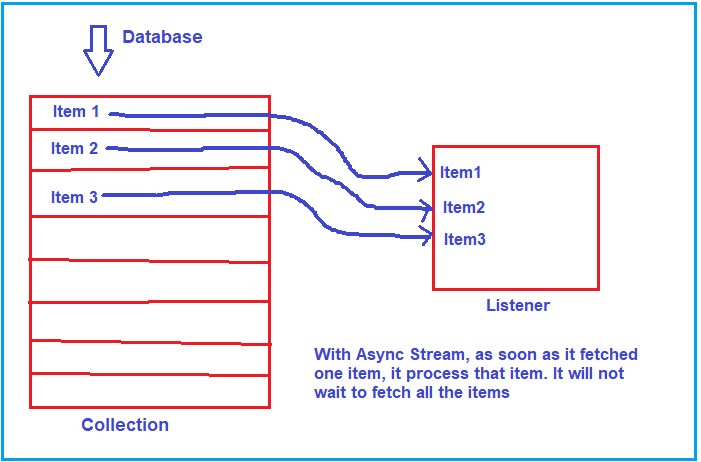
1. **It’s declared with the async modifier.**
2. **It returns an IAsyncEnumerable<T>.**
3. **The method contains yield return statements to return successive elements in the asynchronous stream.**

در async stream نیاز است که قبل از foreach از کلمه کلیدی await استفاده کرد.

امکان ValueTask نیز به عنوان Return type وجود دارد.



در حالت async stream بلافاصله بعد از این که هر یک از ایتم ها داده های ما فراهم شده فراخواننده آن را دریافت می کند مثال 100 تا دیتا از دیتابیس می خوانیم به محض آماده شده هر یک از داده ها فراخواننده آن را دریافت می کند اما در حالت های معمول این طور نیست بلکه باید 100 باهم آماده شده و برگردانده شود.

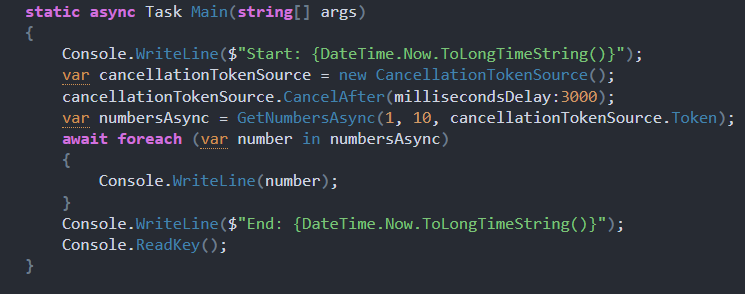


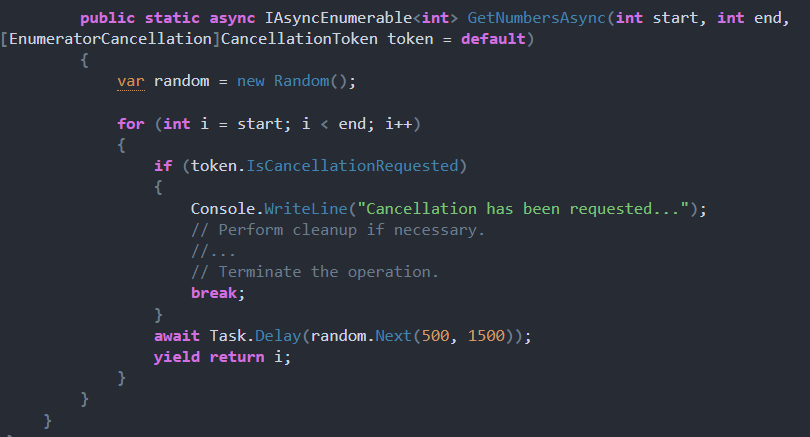
در نهایت سبب عملکرد بهتر برنامه و تجربه کاربری بهتری می شود.



**Cancellation Token in C#:**

برای کنسل کردن عملیات کاربرد دارد. به طور مثال بعد 3000 میلی ثانیه می خواهیم فرآیند اجرا متوقف شود. در ادامه مثال مشاده خواهید کرد.





**ConfigureAwait Method in C#:**

برای مشخص کردن این که چگونه await متظر دریافت Task باشد. مقدار True برای دریافت مقدار در همان context و در غیر این صورت مقدار false

var numbersAsync =

GetNumbersAsync**(**1, 10, cancellationTokenSource.Token**)**.ConfigureAwait**(false)**;

**await** **foreach** **(**var number in numbersAsync**)**

**{**

Console.WriteLine**(**number**)**;

**}**

**Which do I use, ConfigureAwait True or False?**

The direct answer to this question is:

1. If you are writing code for the UI, use ConfigureAwait(true).
2. If you are writing code in a library that will be shared, use ConfigureAwait(false)