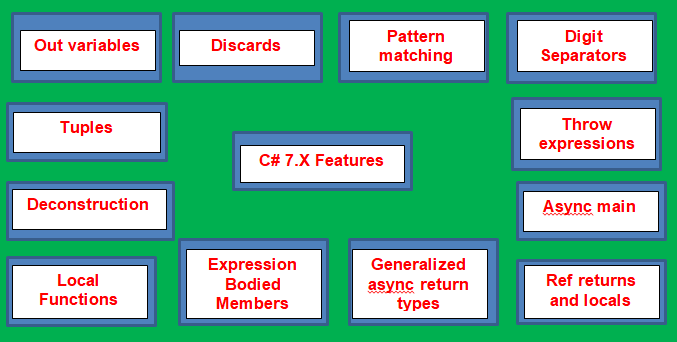
**C# 7 New Features**

**C# 7 New Features**

In this article, I am going to discuss **C# 7 New Features**with examples. The C# 7 comes with a number of new features which brings a focus on data consumption, code simplification and performance.

I think the biggest C# 7 new features are tuples and pattern matching. Now you can return multiple different or same type of values from a method using tuples. The pattern matching is the concept which is used to simplifies the code that is conditional on the shape of data. Along with these two features, there are also many other features are also introduced. In this article, we are going to discuss what are the new features introduced in C# 7. Please have a look at the following diagram.



**Following are the new features that are introduced as part of C# 7**

1. [**Out variables and Discards**](https://dotnettutorials.net/lesson/out-variables-csharp7/)
2. [**Pattern matching**](https://dotnettutorials.net/lesson/pattern-matching-csharp/)
3. [**Digit Separators**](https://dotnettutorials.net/lesson/digit-separators-csharp/)
4. [**Tuples**](https://dotnettutorials.net/lesson/tuples-csharp/)
5. [**Deconstruction (Splitting Tuples)**](https://dotnettutorials.net/lesson/splitting-tuples-csharp/)
6. [**Local functions**](https://dotnettutorials.net/lesson/local-functions-csharp/)
7. [**Ref returns and Ref locals**](https://dotnettutorials.net/lesson/ref-locals-and-ref-returns-chsarp/)
8. [**Generalized async return types**](https://dotnettutorials.net/lesson/generalized-async-return-types/)
9. [**Expression bodied members**](https://dotnettutorials.net/lesson/expression-bodied-members-csharp/)
10. [**Throw expressions**](https://dotnettutorials.net/lesson/thrown-expressions-csharp/)
11. [**Async main**](https://dotnettutorials.net/lesson/async-main-csharp/)

We are going to discuss each of these new features in details with examples from our next articles. In the next article, I am going to discuss the [**Out variables in C# 7**](https://dotnettutorials.net/lesson/out-variables-csharp7/)with examples.

The most important point that you need to keep in mind is we are going to use **Visual Studio 2017** with **Dot Net framework 4.7.x**. Please be ready with visual studio 2017.

# Out Variables in C#

## ****Out Variables in C# 7 with examples****

In this article, I am going to discuss the improvement of **Out variables in C#** with Examples. With the introduction of C# 7, now it is possible to define the method’s out parameters directly within the method. As part of this article, we are going to discuss the following pointers.

1. **How to work with the Out Parameter Before C# 7?**
2. **Working with Out Variable in C# 7.**
3. **What is the Scope of Out Parameter?**
4. **Can we declare the out variable with var data type from C# 7?**
5. **How to Ignore an out parameter in C#?**
6. **Out Parameter Using TryParse**

##### ****How to work with the Out Parameter Before C# 7?****

In C#, we generally use the **out parameter**to pass a method argument’s reference. If you want to use an **out** parameter, then you need to explicitly specify the **out** keyword in both the calling method and method definition. Before C# 7, we need to split their declaration and usage into two parts i.e. first we need to declare a variable and then we need to pass that variable to the method using the **out** keyword. The **Out** Parameter in C# never carries value into the method definition. So, it is not required to initialize the out parameter while declaring.

##### ****Example: Out Parameter Before C# 7****

Let us understand how to use Out Parameter before C# 7 with an example. Please have a look at the below example. As you can see the GetEmployeeDetails method is created with four out parameters. Then within the Main method, first we declare four variables without initializing. Initialization is optional. Then while calling the GetEmployeeDetails method, we pass the four variables by specifying the out keyword.

**class** Program

**{**

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

**{**

string EmployeeName, Gender, Department;

**long** Salary;

GetEmployeeDetails**(**out EmployeeName, out Gender, out Salary, out Department**)**;

Console.WriteLine**(**"Employee Details:"**)**;

Console.WriteLine**(**"Name: {0}, Gender: {1}, Salary: {2}, Department: {3}",

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

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**static** **void** GetEmployeeDetails**(**out string EmployeeName, out string Gender, out **long** Salary, out string Department**)**

**{**

EmployeeName = "Pranaya Rout";

Gender = "Male";

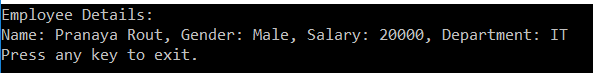
Salary = 20000;

Department = "IT";

**}**

**}**

###### **Output:**



**Note:** The Out Parameter in C# never carries the value into the method definition. So, it is mandatory to initialize the out variables within the method definition otherwise you will get a compile-time error in C#. Again you cannot use the “**var**” data type to declare these variables.

Now, the question is, it is not required to initialize the out variables then why should we split their usage into two parts. Well, this will be overcome with C# 7.

##### ****Working with Out Variable in C# 7.****

With the introduction of C# 7, now it is possible to define a method’s out parameters directly within the method. So the above program can be rewritten as shown below and also gives the same output. Here, you can see, we are directly declaring the variable at the time of method call i.e. GetEmployeeDetails(out string EmployeeName, out string Gender, out long Salary, out string Department);. This will eliminate the need to split the usage of C# out variable in two parts.

**class** Program

**{**

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

**{**

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

Console.WriteLine**(**"Employee Details:"**)**;

Console.WriteLine**(**"Name: {0}, Gender: {1}, Salary: {2}, Department: {3}",

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

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**static** **void** GetEmployeeDetails**(**out string EmployeeName, out string Gender, out **long** Salary, out string Department**)**

**{**

EmployeeName = "Pranaya Rout";

Gender = "Male";

Salary = 20000;

Department = "IT";

**}**

**}**

Run the application and you will get the output as expected as our previous program.

##### ****What is the Scope of Out Parameter in C#?****

As we declared the out parameter directly within the method call, so we need to understand the scope of the out parameter in C#. In the above program, the out variables are in the scope of the enclosing block. So the subsequent line can use them.

##### ****Can we declare the out variable with var data type from C# 7?****

Yes, you can. As the **out** variables are declared directly as arguments to the out parameters, so, the compiler can easily tell what their data type should be. So it is always better to use the “**var**” data type to declare them as shown in the following example.

**class** Program

**{**

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

**{**

GetEmployeeDetails**(**out var EmployeeName, out var Gender, out var Salary, out var Department**)**;

Console.WriteLine**(**"Employee Details:"**)**;

Console.WriteLine**(**"Name: {0}, Gender: {1}, Salary: {2}, Department: {3}",

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

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**static** **void** GetEmployeeDetails**(**out string EmployeeName, out string Gender, out **long** Salary, out string Department**)**

**{**

EmployeeName = "Pranaya Rout";

Gender = "Male";

Salary = 20000;

Department = "IT";

**}**

**}**

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

If you want to ignore an out parameter then you need to use a wildcard called **underscore (‘\_’)** as the name of the parameter. For example, if you don’t care about the **Department** parameter, then you just replace it with an **underscore (‘\_’)** as shown below.

**class** Program

**{**

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

**{**

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

Console.WriteLine**(**"Employee Details:"**)**;

Console.WriteLine**(**"Name: {0}, Gender: {1}, Salary: {2}",

EmployeeName, Gender, Salary**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**static** **void** GetEmployeeDetails**(**out string EmployeeName, out string Gender, out **long** Salary, out string Department**)**

**{**

EmployeeName = "Pranaya Rout";

Gender = "Male";

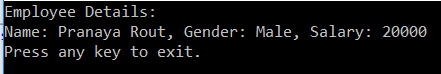
Salary = 20000;

Department = "IT";

**}**

**}**

**OUTPUT:**



##### ****Out Parameter Using TryParse in C#****

When we are working with real-time applications, then the common use of the **out** variable is the Try… pattern, where a boolean return value indicates the success, and if successful then the **out** parameters carry the results. Let us understand this with an example.

##### ****Example: Try Pattern using Out variable Before C# 7****

Let us first see an example of using C# out variable with try pattern before C# 7. Please have a look at the following example. In the below example, first, we declare and initialize a string variable and then we declare a DateTime variable. Then within the if block we are calling the DateTime.TryParse and passing the first parameter as the string variable and the second one is the out data time parameter. If the above string is converted to DateTime, then DateTime.TryParse method will return true and the converted value will be stored in the out variable in C#.

**class** Program

**{**

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

**{**

string s = "09-Jun-2018";

DateTime date;

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

**{**

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

**}**

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

###### **Output:**

C# out Parameters

##### ****Example: Try Pattern using Out variable From C# 7****

With the introduction of C# 7, the previous example can be rewritten as shown below. As you can see, now we don’t require to split the usage of the out variable into two parts. Directly we can declare the out variable within the method itself.

**class** Program

**{**

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

**{**

string s = "09-Jun-2018";

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

**{**

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

**}**

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

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

Run the application and it should give the same output as the previous example. In the above program, we are declaring the **out** variable within the method and it is being accessed from outside also. If an exception occurred, then the **out** variable will be assigned with a default value. Let’s see this with an example.

**class** Program

**{**

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

**{**

string s = "09-Junnnneee-2018";

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

**{**

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

**}**

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

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

###### **Output:**

Out Parameters Example in C#

**Pattern Matching in C#**

**Pattern Matching in C# with Examples**

In this article, I am going to discuss **Pattern Matching in C#** with Examples. Please read our previous article where we discussed the improvement of [**Out Variables in C#**](https://dotnettutorials.net/lesson/out-variables-csharp7/) with examples. **Pattern Matching** is a new feature that was introduced in C# 7.0. As part of this article, we are going to discuss the following pointers.

1. **What is Pattern Matching in C#?**
2. **How to implement Pattern Matching in C#?**
3. **How to implement Pattern Matching Before C# 7?**
4. **Pattern Matching Using is Expression.**
5. **How to Implement Pattern Matching using a switch statement?**
6. **Understanding the use of  When clauses in the case statement in C#**

**What is Pattern Matching in C#?**

Pattern Matching is a mechanism that tests a value i.e. whether the value has a specific shape or not. If the value is in a specific shape then it will extract the data from the value. If this is not clear at the moment, then don’t worry we will understand this with multiple examples.

**How to implement Pattern Matching in C#?**

To implement **Pattern Matching in C#**, we are provided with two language constructs such as:

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

In the upcoming versions of C#, we may expect more pattern matching expressions. Pattern Matching is useful in many ways however C# 7.0 currently supports the following.

1. It can be used with any data type including the custom data types whereas if/else can only be used with primitive types.
2. Pattern matching has the ability to extract the data from the expression.

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

The **“is”** operator is available from the first version of C# and it is used to check whether an object is compatible with a specific type or not. For example, if a specific interface is implemented, or if the type of the object derives from a base class or not. The result of this operator is true or false. If this is not clear at the moment then don’t worry, we will try to understand this with some examples.

**Example: Pattern Matching in C# with “is” Expression**

Let us understand how to implement Pattern Matching in C# using the “**is**” expression with an example. First, create one class with the name Shape and then create four classes with the names **Shape, Circle, Rectangle, and Tringle**as shown below. The following code is very straightforward. Here, we have created one base class i.e. Shape, and three derived classes i.e Rectangle, Circle, and Triangle, and all these classes are derived from the base Shape class. Further, if you notice, we have created some properties within each derived class and initialize the properties using the class constructor.

**using** *System;*

**namespace** *PatternMatchingDemo*

**{**

**public** **class** Shape

**{**

**public** const **float** PI = 3.14f;

**}**

**public** **class** Circle : Shape

**{**

**public** **double** Radius **{** **get**; **}**

**public** Circle**(double** radius**)**

**{**

Radius = radius;

**}**

**}**

**public** **class** Rectangle : Shape

**{**

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

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

**public** Rectangle**(double** length, **double** height**)**

**{**

Length = length;

Height = height;

**}**

**}**

**public** **class** Triangle : Shape

**{**

**public** **double** Base **{** **get**; **}**

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

**public** Triangle**(double** @**base**, **double** height**)**

**{**

Base = @**base**;

Height = height;

**}**

**}**

**}**

**Pattern Matching Before C# 7:**

Let us first understand our requirements. We need to create a method with one parameter of type Shape. The reason is, the Shape class is the base class and it can hold the object reference of any of its child classes like Rectangle, Triangle, and Circle. Please modify the Program class as shown below. Please have a look at the **DisplayArea()** method. Here in the **DisplayArea()** method, we are testing each type in a series of **“if”** and **“is”** statements and then we are explicitly casting the type to a specific type and then doing some action.

**using** *System;*

**namespace** *PatternMatchingDemo*

**{**

**class** Program

**{**

**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**)**

**{**

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

**{**

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

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

**}**

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

**{**

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

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

**}**

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

**{**

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**))**;

**}**

**}**

**}**

**}**

Now, let us understand how to use the new Pattern Matching Mechanism which was introduced in C# 7.0.

**Pattern Matching Using is Expression:**

We can simplify the previous example by using the “**is**” expression pattern which will check and assign the value to a variable. So, in order to do this, please modify the **DisplayArea()**method of the Program class as shown below.

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

**{**

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

**{**

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

**}**

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

**{**

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

**}**

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

**{**

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

**}**

**else**

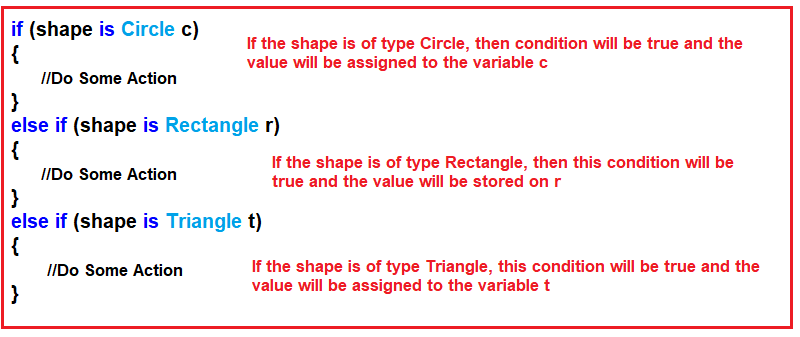
**{**

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

**}**

**}**

In the above example, we are using the **“is”** expressions which will test the variable type and if it matches the type then it assigns that value to the variable. For better understanding please have a look at the following image.



**Pattern Matching using the switch statement in C#**:

The traditional switch statement in C# is also a pattern-matching expression. Let us see how to use the switch statement to implement the previous example. Modify the DisplayArea method as shown below to implement Pattern Matching using a switch statement in C#.

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

**{**

**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**))**;

**}**

**}**

**Points to Remember while working with Case Statement in C#:**

You need to remember the following points while working with the newly extended switch statement for Pattern Matching in C#.

1. **The default clause is always evaluated last:**In our example, the **null case** statement comes at the last but it will be checked before the default case statement is checked. The reason for this is for the compatibility with the existing switch statements. So it is always advisable and a good programming practice to put the default statement at the end.
2. **The order of case clauses is now mattered:** Just like the catch clauses in the try block, the first one that matches in the case statement gets picked. So as a developer it is important to write the case statement in the proper order.

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

Let us understand the use of case Expression using the when clause in C# with an example. In our example, when the length and height both are the same for the rectangle then we need to treat it as a Square and display the message accordingly. We can specify this condition using the when clause. So, modify the main method and DisplayArea method of the Program class as shown below.

**namespace** *PatternMatchingDemo*

**{**

**class** Program

**{**

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

**{**

Rectangle square = new Rectangle**(**10, 10**)**;

DisplayArea**(**square**)**;

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

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

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

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

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

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

Console.ReadKey**()**;

**}**

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

**{**

**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**))**;

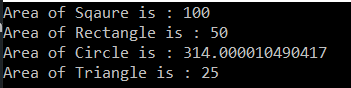
**}**

**}**

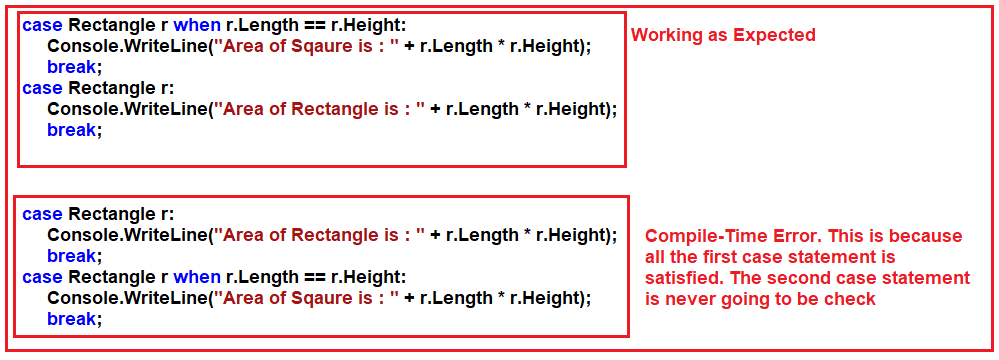
**}**

**}**

Now, run the application and you should the output as shown below.



The most important point that you need to remember is, you need to place the case statement with the when clause first then the normal case statement for the same type. Please have a look at the following diagram for a better understanding.



**Digit Separators in C#**

**Digit Separators in C# with examples**

In this article, I am going to discuss the **Digit Separators in C#** with Examples. Please read our previous article where we discussed [**Pattern Matching in C#**](https://dotnettutorials.net/lesson/pattern-matching-csharp/) with Examples. At the end of this article, you will understand what Digit Separator is in C# and when and how to use Digit Separators with Examples.

**What is Digit Separator in C#?**

In reality, it’s very difficult to read a very large number. To overcome this problem, C# 7 comes with a new feature called digit separators “**\_”**. Now, it is possible to use one or more Underscore **(\_)** characters as digit separators in C# to represent a very big number. Sometimes, it is required when we are going to represent a very big number.

**Example: Digit Separator in C#**

Let’s understand Digit Separators in C# with an example. Please have a look at the below example. As you can see, here we have created two variables. The first variable is holding a long value while the second variable is also holding a long value but that value is separated by underscores. If you look at the value, both are the same. But the readability is better in the case of the second variable which is split by underscores.

**class** Program

**{**

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

**{**

// Both are equivalent.

var bigNumber = 123456789012345678;

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

Console.WriteLine**(**"bigNumber : {0}, bigNumberSplit : {1}", bigNumber, bigNumberSplit**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

When we run the application it gives you the following output.

Digit Separators in C# 7

If you observe in the code, then you feel a little difficult to read the first number as it is a very big number. But, you can easily read the second number because of the thousands separator. The separators do not make any difference in the value as you can see in the above output. You can place them wherever you like in the number, and in any quantity.

**Example: Digit Separator with double**

It is also not mandatory to use a single underscore as a separator even though you can also use multiple separators. And it is not restricted to using them with integers only; they also work with the other numeric types like double, float as well. So, let us see an example, where we will use more than one underscores as digit separators as well as working with double value. Please have a look at the following example. Modify the main method as shown below.

**class** Program

**{**

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

**{**

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

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

Console.WriteLine**(**"bigNumber : {0}, bigNumberSplit : {1}", myData1, myData2**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

Now when we run the application, it gives us the output as expected as shown in the below image.

Digit Separators in C#

**Tuples in C#**

**Tuples in C# with Examples**

In this article, I am going to discuss the **Tuples in C#** with Examples. Please read our previous article where we discussed the [**Digit Separators in C#**](https://dotnettutorials.net/lesson/digit-separators-csharp/) with examples. As part of this article, we are going to discuss the following pointers.

1. **Why do we need Tuples in C#?**
2. **What are the different ways to return more than one value from a method?**
3. **Examples of Tuples Before C# 7.**
4. **Understanding the Problems with the Tuples Before C# 7.**
5. **How to use Tuples from C# 7.**
6. **Tuples in C# with named Parameters**
7. **Guidelines to use Tuples**

**Why do we need Tuples in C#?**

If you want to return more than one value from a method then you need to use **Tuples in C#**. And in the programming world, it is a very common thing to return multiple values from a method. Tuples are not completely new in C# 7. In .NET Framework 4.0, a set of **Tuple** classes has been introduced in the **System** namespace. Tuples in C# 7, provide a better mechanism to return multiple values from a method.

**What are the different ways to return more than one value from a method in C#?**

Following are the different mechanisms available in C# to return multiple values from a method:

1. **Using Custom DataType:** You can return multiple values from a method by using a **custom data type (i.e. class)** as the return type of the method. But sometimes we don’t need or don’t want to use classes and objects because that’s just too much for the given purpose.
2. **Using Ref and Out variable:** You can also return more than one value from the method either by using the **“out”** or **“ref”** parameters. Using **“out”** or **“ref”** parameters is quite difficult to understand and moreover, the **“out” and “ref”** parameters will not work with the async methods.
3. **Using dynamic keyword:** You can also return multiple values from a method by using the dynamic keyword as the return type. The **dynamic** keyword was introduced in C# 4. But from a performance point of view, we probably don’t want to use dynamic.

As I already told tuples are not new to C# 7. They come with much more improvements in C# 7. So, let us first understand the Tuples which are there before C# 7, and then we will see what improvements they have done with types in C# 7.

**Tuples Before C# 7:**

In the following example, we are returning two values (integer and double) from the Calculate method using the Tuple class. In the following example, within the calculate method we create an instance of the Tuple class by calling the static Create method of the Tuple class. To the static Create method, we are passing the required integer and double values that we want to return from the method. In the Main method, we are storing the result in a Tuple variable and then accessing the first value i.e. count using the item1 property and the second value using the item2 property.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *TulesDemo*

**{**

**class** Program

**{**

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

**{**

var values = new List**<double>()** **{** 10, 20, 30, 40, 50 **}**;

Tuple**<int**, **double>** t = Calulate**(**values**)**;

Console.WriteLine**(**$"There are {t.Item1} values and their sum is {t.Item2}"**)**;

Console.ReadKey**()**;

**}**

//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;

**}**

**}**

**}**

**Problems with the above code:**

There are 3 major problems in the above code with Tuple:

1. The **first problem** is that the Tuples in C# are classes, i.e. reference types. As reference types, the memory is allocated on the heap area and garbage is collected only when they are no longer used. For applications where performance is a major concern, it can be an issue.
2. The **second problem** is that the elements in the tuple don’t have any names and you can only access them by using the names Item1, Item2, Item3, etc. which are not meaningful at all. The **Tuple<T1, T2>** type does not provide any information about what the tuple actually represents which makes it a poor choice in public APIs.
3. The **third problem** is that you can use a maximum of 8 properties in a Tuple in C#. If you want to return more than 8 values from a method, then again the last argument of the Tuple must be another Tuple. This makes the syntax more difficult to understand.

**How to overcome the above Problems in C#?**

To overcome the above three problems, C# 7 comes with a new feature that is used to improve the support for tuples in C#. With C# 7, now it is possible to declare the tuple as **“inline”**, which is like an anonymous type, except that they are not limited to the current method. Let’s modify the code as shown below to see the use of new improvements of Tuples in C# 7. In your code, if you are getting **Predefined type ‘System.ValueTuple´2´ is not defined or imported** error, then you need to add the **System.ValueTuple** package from NuGet Package Manager.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *TulesDemo*

**{**

**class** Program

**{**

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

**{**

var values = new List**<double>()** **{** 10, 20, 30, 40, 50 **}**;

var result = Calulate**(**values**)**;

Console.WriteLine**(**$"There are {result.Item1} values and their sum is {result.Item2}"**)**;

Console.ReadKey**()**;

**}**

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

**{**

**int** count = 0;

**double** sum = 0.0;

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

**{**

count++;

sum += **value**;

**}**

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

**}**

**}**

**}**

As you can see in the above code, we are returning two values i.e. int and double from the Calculate method and then we are accessing the values. This is much better. If you want then you can also give specific names to the tuples returning values.

**Tuples in C# with named Parameters:**

With C# 7, now it is possible to provide the tuples parameters with user-defined names. To understand this let’s modify the code as shown below. Here we are providing names for the tuple parameters in the Calculate method as count and sum. Now, in the Main method, you can access these parameters and moreover, you will also get intelligence.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *TulesDemo*

**{**

**class** Program

**{**

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

**{**

var values = new List**<double>()** **{** 10, 20, 30, 40, 50 **}**;

var result = Calulate**(**values**)**;

Console.WriteLine**(**$"There are {result.count} values and their sum is {result.sum}"**)**;

Console.ReadKey**()**;

**}**

**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**)**;

**}**

**}**

**}**

**Provide Explicitly names while storing the Result:**

In the following example, we are providing explicitly names to the tuple properties with the Main method where we calling the Calculate method. In this case, you don’t require to provide the variable name as we can access the properties directly with the provided name.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *TulesDemo*

**{**

**class** Program

**{**

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

**{**

var values = new List**<double>()** **{** 10, 20, 30, 40, 50 **}**;

var**(**countResult, SumResult**)** = Calulate**(**values**)**;

Console.WriteLine**(**$"There are {countResult} values and their sum is {SumResult}"**)**;

Console.ReadKey**()**;

**}**

**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**)**;

**}**

**}**

**}**

**Guidelines to use Tuples:**

Basically, one and the most important question that comes to our mind is when to use Tuples and when to use Classes to return more than one value from a method in C#. The answer is it depends on the business requirement. However, there are some guidelines and rules that you need to follow which will guide you to choose between them:

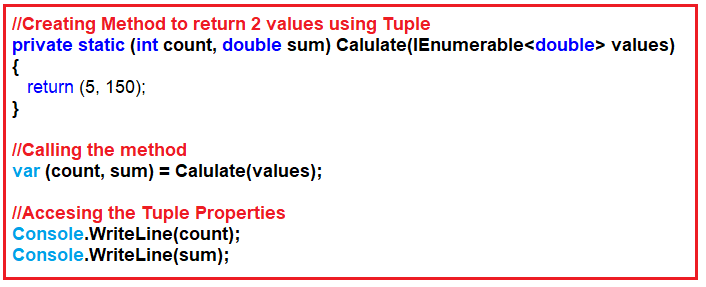
**Tuples in C# 7 are values, so they are copied by value, rather than by reference.**

Most of the time, this should not be an issue. However, if you are passing around tuples of large structs, this might have an impact on the performance of the application. Ref locals/returns can be used to work around these performance issues that we will discuss in our upcoming articles. As the tuples in C# 7 are values, so modifying a copy will not change the original copy.

**Just use common sense**

For any situation where you might consider using a tuple: simply ask yourself the question: “**will a tuple simplify the code here**“. If the answer is “**yes**“, then use a tuple. And that ultimately is the primary consideration over whether to use a tuple or a custom class.

So in simple words, we can say that a Tuple is an ordered sequence of heterogeneous objects. The Tuples in C# are going to be used when a method is going to return more than one value.



# Splitting Tuples in C#

## ****Splitting Tuples in C# with Examples****

In this article, I am going to discuss **Splitting Tuples in C#** with Examples. Please read our previous article before proceeding to this article where we discussed [**Tuples in C#**](https://dotnettutorials.net/lesson/tuples-csharp/) with examples. The Splitting Tuples in C# is a process of splitting a variable value into multiple parts and storing each part into a new variable. This is very useful when you are working with tuples in C# as we know Tuples are going to store multiple values.

##### ****Why do we need to Split Tuples in C#?****

As we already discussed Tuples provides a lightweight way to retrieve multiple values from a method call. Once you retrieve the tuple, then you need to handle its individual elements. Handling this element one by one is really a dirty approach. We can overcome this by splitting the tuples in C#.

##### ****Example: Splitting Tuples in C#****

Let’s understand Splitting Tuples in C# with an example. Please have a look at the below example. As you can see in the below code, we are using Tuples to return four values from the GetEmployeeDetails method. And further, if you notice within the Main method, we are storing each value of the Tuple in separate variables.

**class** Program

**{**

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

**{**

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

var Name = EmployeeDetails.Item1;

var Salary = EmployeeDetails.Item2;

var Gender = EmployeeDetails.Item3;

var Dept = EmployeeDetails.Item4;

// Do something with the data.

//here we are just printing the data in the console

Console.WriteLine**(**"Employee Details :"**)**;

Console.WriteLine**(**$"Name: {Name}, Gender: {Gender}, Department: {Dept}, Salary:{Salary}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

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

**{**

//based on the EmployyeID get the data from a database

//here we are hardcoded the value

string EmployeeName = "Pranaya";

**double** Salary = 2000;

string Gender = "Male";

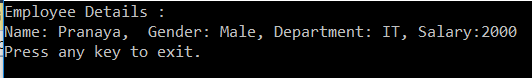
string Department = "IT";

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

**}**

**}**

When you run the application, you will get the data as expected as shown below.



As shown in the above example the **GetEmployeeDetails()** method returns a tuple with 4 values and then we assigned each of its elements to a variable in a separate operation. But from C# 7.0, now we can retrieve multiple elements from a tuple or retrieve multiple fields or properties values from an object in a single operation which is called **Splitting Tuples in C#**.

##### ****Different ways to deconstruct a tuple or Splitting Tuples in C#:****

**Way1:**We can explicitly declare the type of each field inside the parentheses. Let’s modify the program as shown below to understand this concept.

**class** Program

**{**

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

**{**

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

// Do something with the data.

//here we are just printing the data in the console

Console.WriteLine**(**"Employee Details :"**)**;

Console.WriteLine**(**$"Name: {Name}, Gender: {Gender}, Department: {Dept}, Salary:{Salary}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

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

**{**

//based on the EmployyeID 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**)**;

**}**

**}**

The above example deconstructs the 4-tuple returned by the GetEmployeeDetails() method explicitly by declaring the types of each filed within the parenthesis.

###### **Way2:**

You can also use the var keyword so that C# infers the type of each variable. You can place the var keyword outside of the parentheses. Let us understand this by modifying the code as shown below.

**class** Program

**{**

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

**{**

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

// Do something with the data.

//here we are just printing the data in the console

Console.WriteLine**(**"Employee Details :"**)**;

Console.WriteLine**(**$"Name: {Name}, Gender: {Gender}, Department: {Dept}, Salary:{Salary}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

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

**{**

//based on the EmployyeID 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**)**;

**}**

**}**

The above example uses type inference when deconstructing the 4-tuple returned by the GetEmployeeDetails method. You can also use the var keyword individually with any or all of the variable declarations inside the parentheses. Let us understand this with an example.

**class** Program

**{**

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

**{**

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

// Do something with the data.

//here we are just printing the data in the console

Console.WriteLine**(**"Employee Details :"**)**;

Console.WriteLine**(**$"Name: {Name}, Gender: {Gender}, Department: {Dept}, Salary:{Salary}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

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

**{**

//based on the EmployyeID 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**)**;

**}**

**}**

Note: This method is cumbersome and is not recommended.

###### **Way3:**

You may deconstruct the tuple into variables that have already been declared. Let us understand this with an example

**class** Program

**{**

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

**{**

string Name;

**double** Salary;

string Gender = "Female";

string Dept = "HR";

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

// Do something with the data.

//here we are just printing the data in the console

Console.WriteLine**(**"Employee Details :"**)**;

Console.WriteLine**(**$"Name: {Name}, Gender: {Gender}, Department: {Dept}, Salary:{Salary}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

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

**{**

//based on the EmployyeID 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**)**;

**}**

**}**

###### **Points to Remember while Splitting Tuples in C#:**

Note that you cannot specify a specific type outside the parentheses even if every field in the tuple has the same type. This generates compiler error **CS8136, “Deconstruction ‘var (…)’ form disallows a specific type for ‘var’.”.**

Note that you must assign each element of the tuple to a variable. If you omit any elements, the compiler generates error **CS8132, “Cannot deconstruct a tuple of ‘x’ elements into ‘y’ variables.”**

You cannot mix declarations and assignments to existing variables on the left-hand side of a deconstruction. The compiler generates error **CS8184, “a deconstruction cannot mix declarations and expressions on the left-hand side.”** when the members include newly declared and existing variables.

**Local Functions in C#**

**Local Functions in C# with Examples**

In this article, I am going to discuss the **Local Functions in C#**with Examples that are introduced as part of C# 7. Please read our previous article before proceeding to this article where we discussed how to [**split a tuple in C#**](https://dotnettutorials.net/lesson/splitting-tuples-csharp/) with examples. The Local Functions means a function is declared and defined inside another function.

**What are Local Functions in C#?**

The **Local Functions in C#**are the special kind of inner function or you can say sub-function or function within a function that can be declared and defined by the parent function. These methods or functions are the private methods for their containing type and are only called by their parent method.

**Why do we need Local Functions in C#?**

If you want to execute some piece of code multiple times within a method then you can put those codes as an inner function or you can say local function within that method. Then call that local function whenever required from the parent method. Some of the examples where we can create local functions are as follows

1. Small helper functions to be used several times within the main or parent method.
2. Parameter validation functions for any iterators or asynchronous methods.
3. An alternate to recursive functions as local function comparatively takes less memory due to the reduced call stack.

**Example: Local Functions in C#**

Let’s understand Local Functions in C# with one example. Please have a look at the below code. As you can see, within the Main method we are defining two inner methods i.e. Sum and Difference. The Main method is called the Parent Method and the Sum and Difference methods are called Local Function or Methods. You can access Sum and Difference method in the context of the Main method only.

**class** Program

**{**

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

**{**

**int** a = 10, b = 5;

**int** sum = Sum**(**a, b**)**;

**int** difference = Difference**(**a, b**)**;

Console.WriteLine**(**$"The Sum of {a} and {b} is {sum}"**)**;

Console.WriteLine**(**$"The Difference of {a} and {b} is {difference}"**)**;

**int** Sum**(int** x, **int** y**)**

**{**

**return** x + y;

**}**

**int** Difference**(int** x, **int** y**)**

**{**

**return** x - y;

**}**

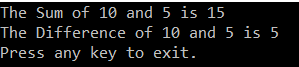
Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**}**

When we run the application, it will give us the following output.



As you can see in the above example, we have created two nested functions “**Sum**” and “**Difference**”. These two local functions can be called from anywhere by the parent’s main function only.

**Points to Remember while working with Local Functions:**

The following points you need to keep in mind while working with the Local Functions.

1. You can not overload a Local Function in C#
2. The Accessibility modifiers such as public, private, 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

**Real-Time Example of Local Functions in C#.**

Let us understand the use of Local Functions with one real-time example. The Parameter validation scenario. In the following example, the**IsRequestValid** local function is used to validate the parameters of the **AddEmployee** function.

**using** *System;*

**using** *System.Text;*

**namespace** *LocalFunctioDemo*

**{**

**class** Program

**{**

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

**{**

Employee employe1 = new Employee**()**

**{**

Id = 1001,

Name = "Pranaya",

Gender = "Male",

Salary = 1000,

Department = "IT"

**}**;

**bool** IsInserted = AddEmployee**(**employe1**)**;

Console.WriteLine**(**$"Is Employee with id 1001 inserted: {IsInserted}"**)**;

Employee employee2 = new Employee**()**

**{**

Id = 1001,

Name = "Pranaya",

Department = "IT"

**}**;

IsInserted = AddEmployee**(**employee2**)**;

Console.WriteLine**(**$"Is Employee with id 1002 inserted: {IsInserted}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

Console.ReadKey**()**;

**}**

**public** **static** **bool** AddEmployee**(**Employee request**)**

**{**

var validationResult = IsRequestValid**()**;

**if** **(**validationResult.isValid == **false)**

**{**

Console.Write**(**$"{ nameof(validationResult.errorMessage)} : { validationResult.errorMessage}"**)**;

**return** **false**;

**}**

// Some code for inserting the Employee in database.

**return** **true**;

**(bool** isValid, string errorMessage**)** IsRequestValid**()**

**{**

**if** **(**request == **null)**

**{**

**throw** new ArgumentNullException**(nameof(**request**)**, $"The { nameof(request) } can not be null."**)**;

**}**

var lsb = new Lazy**<**StringBuilder**>()**;

**if** **(**string.IsNullOrWhiteSpace**(**request.Name**))**

**{**

lsb.Value.AppendLine**(**$"The {nameof(request)}’s {nameof(request.Name)} property can not be empty."**)**;

**}**

**if** **(**string.IsNullOrWhiteSpace**(**request.Gender**))**

**{**

lsb.Value.AppendLine**(**$"The {nameof(request)}’s {nameof(request.Gender)} property can not be empty."**)**;

**}**

**if** **(**string.IsNullOrWhiteSpace**(**request.Department**))**

**{**

lsb.Value.AppendLine**(**$"The {nameof(request)}’s {nameof(request.Department)} property can not be empty."**)**;

**}**

**if** **(**request.Id **<**= 0**)**

**{**

lsb.Value.AppendLine**(**$"The {nameof(request)}’s {nameof(request.Id)} property can not be less than zero."**)**;

**}**

**if** **(**request.Salary **<**= 0**)**

**{**

lsb.Value.AppendLine**(**$"The {nameof(request)}’s {nameof(request.Salary)} property can not be less than zero."**)**;

**}**

**if** **(**lsb.IsValueCreated**)**

**{**

var errorMessage = lsb.Value.ToString**()**;

**return** **(**isValid: **false**, errorMessage: errorMessage**)**;

**}**

**return** **(**isValid: **true**, errorMessage: string.Empty**)**;

**}**

**}**

**}**

**public** **class** Employee

**{**

**public** **long** Id **{** **get**; **set**; **}**

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

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

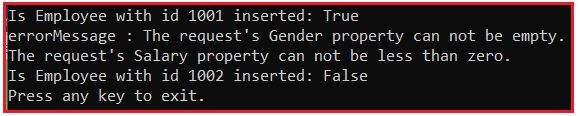
**public** **double** Salary **{** **get**; **set**; **}**

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

**}**

**}**

When we run the application, it will give us the following output.



# Ref Returns and Ref Locals in C#

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

In this article, I am going to discuss **how to use Ref Returns and Ref Locals in C#** with examples. These two new features are introduced as part of C# 7. Please read our previous article where we discussed how to work with the [**Local Functions in C#**](https://dotnettutorials.net/lesson/local-functions-csharp/) with examples. As of now, we have used ref as the method parameter. But from C# 7, now we can use the ref keyword as the return type of method as well as we can also create local variables using the ref keyword. Before understanding these two new features (**Ref Returns and Ref Locals in C#**) let’s have a look at how we can pass the data and reference as of now.

1. **Passing by Value**
2. **Passing by Reference**
3. **Out parameter**

Let’s discuss these concepts first by taking some simple examples before understanding the ref returns and ref locals in c#.

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

If we declare a parameter of a value type, then the value is allocated within the scope of the method and destroyed immediately at the end of the method execution. Thus, the changes to the value will not affect the caller of the method.

##### ****Example: Pass by Value in C#****

Let us understand Pass by Value in C# with an example. Please have a look at the below example. In the below example, the method **PassByValue(int x)** receives a copy of a value type. The variable **“x”** is allocated within the method. While invoking this method we passed a copy of the variable **“no”** not the actual variable. So the change to **“x”** only happens within the scope of the **PassByValue()**method which will not affect the caller of the method.

**class** Program

**{**

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

**{**

**int** no = 1;

PassByValue**(**no**)**;

Console.WriteLine**(**$"After the invocation of {nameof(PassByValue)} method, {nameof(no)} = {no}"**)**;

Console.WriteLine**(**"Press any key to Exit."**)**;

Console.ReadLine**()**;

**}**

**static** **void** PassByValue**(int** x**)**

**{**

x = 2;

**}**

**}**

When we run the application, it will give us the following output.

Ref Locals and Ref Returns in C#

As you can see in the above image, it will print the value as 1 because a copy of the value is passed, the value of “no” never changes, and thus after the invocation of the PassByValue() method, the variable no remains unchanged.

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

If you want to return the changed data from a method, then you need to use a return type for that method. But in C#, there are some other ways to return values from a method. One of the ways is to declare a parameter with the **ref** modifier. This allows for returning not only one value but multiple values from a method in C#. With the **ref** parameters in C#, the method can receive and return a value.

##### ****Example: Pass by Reference in C#****

Let us understand Pass by Reference in C# with an example. Please have a look at the below example. In the below example, the parameter **x** of the **PassByReference()** method has been assigned with the **ref** modifier. This **“ref”** modifier in C# indicates that the method invocation is going to happen by using a pointer. The variable **x** references the allocated value of the variable “**no**“. Thus changing the value to **2**, the variable **“no”** now contains the changed value after the invocation of the PassByReference() method.

**class** Program

**{**

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

**{**

**int** no = 1;

PassByReference**(ref** no**)**;

Console.WriteLine**(**$"After the invocation of {nameof(PassByReference)} method, {nameof(no)} = {no}"**)**;

Console.WriteLine**(**"Press any key to Exit."**)**;

Console.ReadLine**()**;

**}**

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

**{**

x = 2;

**}**

**}**

###### **Output:**

Ref Locals and Ref Returns in C#

With the change, the result on the output console is after the invocation of PassByReference, no = 2. The point that you need to remember is in the case of ref, the variable must be initialized before sending to the calling method.

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

As we already discussed with the help of the **ref**modifier in C#, a value can be passed to a method, and also the value can be returned from the method. But, if we have a scenario where we need to only return the value from the method, then in such scenarios we need to use the **out**modifier in C#.

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

Let us understand Out Parameter in C# with an example. Please have a look at the below example. In the below example, in order to call the Out Method, the variable no that is passed as a parameter to the Out Method does not need to be initialized. It is initialized within the Out Method. Here, initialization is optional, as it is mandatory to be initialized in the Out Method body, else you will get a compile-time error.

**class** Program

**{**

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

**{**

**int** no;

OUT**(**out no**)**;

Console.WriteLine**(**$"After the invocation of OUT method, no = {no}"**)**;

Console.WriteLine**(**"Press any key to Exit."**)**;

Console.ReadLine**()**;

**}**

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

**{**

x = 2;

**}**

**}**

###### **Output:**

Ref Locals and Ref Returns in C#

When we run the application, the value returned from the OutMethod is shown, after the invocation of the OutMethod, no = 2. With C# 7.0, it offers a shorter syntax for invoking the method without parameters. The variable can be declared directly within the invocation.

##### ****Example: Out Variable Declare within Method call in C#****

Let us understand How to Declare Out Parameter within Method call in C# with an example. Please have a look at the below example. Here, we are declaring the out variable number directly at the time of the method call.

**class** Program

**{**

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

**{**

OUT**(**out **int** no**)**;

Console.WriteLine**(**$"After the invocation of OUT, no = {no}"**)**;

Console.WriteLine**(**"Press any key to Exit."**)**;

Console.ReadLine**()**;

**}**

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

**{**

x = 2;

**}**

**}**

###### **Output:**

Ref Locals and Ref Returns in C#

#### ****Difference between Ref and Out Parameters in C#****

###### **Ref:**

1. The parameter or argument must be initialized first before it is being passed to ref.
2. It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method.
3. Passing a parameter value by Ref is useful when the called method is also needed to modify the passed parameter.
4. It is not compulsory to initialize a parameter value before using it in a calling method.
5. The ref tells the compiler that the object is initialized before entering the function. so the data can be passed bi-directionally.
6. When you want to pass the value as well as you want to return the modified value then you need to use ref.

##### ****Out:****

1. It is not compulsory to initialize a parameter or argument before it is passed to an out.
2. A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.
3. Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
4. A parameter value must be initialized within the calling method before its use.
5. The out tells the compiler that the object will be initialized inside the function, so the data is passed only in a unidirectional way i.e. from the called method to the caller method.
6. When you only want to return the value from the method then you need to use the out parameter.

**Note:** The point that you need to keep in mind is if a method just returns one value, then it’s always better to use a return type instead of the out or ref modifier. OK. That’s cool. Let’s move to our main topic that is Ref local and Ref return in C# which was introduced as part of C# 7.

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

The **Ref local in C#** is a new variable type that is used to store the references. It is mostly used in conjunction with Ref returns to store the reference in a local variable. That means Local variables now can also be declared with the ref modifier. If this is not clear at the moment, then don’t worry, we will try to understand with some examples.

##### ****Example: Ref local in C#****

Let us understand Ref local in C# with an example. Please have a look at the below example. In the below example, first, we create an integer variable called no1 and initialized it with the value 1. Then we create another integer variable with the ref keyword with the name no2 and initialized it with the reference of no1 i.e. ref int no2 = ref no1; Now, the variable no2 references variable no1, and thus changing no2 changes no1 as well.

**class** Program

**{**

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

**{**

**int** no1 = 1;

**ref** **int** no2 = **ref** no1;

no2 = 2;

Console.WriteLine**(**$"local variable {nameof(no1)} after the change: {no1}"**)**;

Console.WriteLine**(**"Press any key to Exit."**)**;

Console.ReadLine**()**;

**}**

**}**

###### **Output:**

Ref Locals and Ref Returns in C#

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

As a developer, you may be aware of the C# “ref” keyword and its behaviors. Before C# 7.0, the ref was only used to be passed as a parameter in a method, however, there was no provision to return it and use it later. With C# 7.0, this constraint has been waived off and now you can return references from a method as well. This change is really adding flexibility to handle the scenarios when we want references to return in order to make an in-lined replacement. If this is not clear at the moment, then don’t worry we will try to understand this concept with some examples.

##### ****Example: Ref Returns in C#****

Let us understand Ref Returns in C# with an example. Please have a look at the below example. In the below example, I am searching for an odd number inside an integer array and if it is not found throwing an exception, the method is not returning it as the value but as a reference. So, we need to store that value that has been returned as a reference. To store it in a local variable, we can use the ‘ref’ keyword with local variables, known as ref locals in C#.

**class** Program

**{**

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

**{**

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

**{**

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

**{**

**return** **ref** numbers**[**i**]**; //returning as reference

**}**

**}**

**throw** new Exception**(**"odd number not found"**)**;

**}**

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

**{**

Program p = new Program**()**;

**int[]** x = **{** 2, 4, 62, 54, 33, 55, 66, 71, 92 **}**;

**ref** **int** oddNum = **ref** p.GetFirstOddNumber**(**x**)**; //storing as reference

Console.WriteLine**(**$"\t\t{oddNum}"**)**;

oddNum = 35;

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

**{**

Console.Write**(**$"{x[i]}\t"**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

If you print “oddNum” the first time, then it will print 33 but after that, I have re-assigned its value and set “oddNum =35” now iterating the array and printing elements of the array and you can see that whatever I have done, a modification for “oddNum” from outside is also reflecting inside the array and internal value has been modified from 33 to 35.

###### **Output:**

Ref Returns and Ref Locals in C#

# Generalized Async Return Types in C#

## ****Generalized Async Return Types in C# with Examples****

In this article, I am going to discuss the **Generalized Async Return Types in C#** with examples. Please read our previous article where we discuss [**ref locals and ref returns in C#**](https://dotnettutorials.net/lesson/ref-locals-and-ref-returns-chsarp/) with examples. Before understanding the generalized async return types in C#, let’s have a look at asynchronous programming and try to understand how it works.

If you have worked with the async methods, then you may know the async methods can have the following return types:

1. **Task<TResult>**, this return type is used when the async method returns a value.
2. **Task**, this return type is used when the async method does not return any value.
3. **void,** this return type is used for an event handler.

Let us discussed each of these return types with examples.

##### ****The async method returning Task<T> in C#****

We need to use the **Task<TResult>** return type when the async method is going to return a value after the execution of the method using a return statement. In the following example, the **GetLeisureHours()**async method returns an integer value by using the return statement. So, we specify the **GetLeisureHours()** async method return type as **Task<int>**.

The **ShowTodaysInfo()** async method is going to return a string. So, the return type of this async method is **Task<string>**. One more point that you need to remember is whenever you want to call an async method from another async method then you need to use the **await** keyword while calling the method. In our example, we are calling the **GetLeisureHours()** async method from the **ShowTodaysInfo()** async method and you can see while the **GetLeisureHours()** async method we use the **await** keyword. The FromResult async method is a placeholder for an operation that returns a string. The complete example is given below.

**public** **class** Example

**{**

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

**{**

Console.WriteLine**(**ShowTodaysInfo**()**.Result**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**private** **static** **async** Task**<**string**>** ShowTodaysInfo**()**

**{**

string ret = $"Today is {DateTime.Today:D}\n" +

"Today's hours of leisure: " +

$"{await GetLeisureHours()}";

**return** ret;

**}**

**static** **async** Task**<int>** GetLeisureHours**()**

**{**

// Task.FromResult is a placeholder for actual work that returns a string.

var today = **await** Task.FromResult**<**string**>(**DateTime.Now.DayOfWeek.ToString**())**;

// The method then can process the result in some way.

**int** leisureHours;

**if** **(**today.First**()** == 'S'**)**

leisureHours = 16;

**else**

leisureHours = 5;

**return** leisureHours;

**}**

**}**

###### **Output:**

Generalized Async Return Types in C#

For a better understanding of how this happens let’s separate the call to GetLeisureHours() async method from the application of await as the following code shows.

**public** **class** Example

**{**

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

**{**

Console.WriteLine**(**ShowTodaysInfo**()**.Result**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**private** **static** **async** Task**<**string**>** ShowTodaysInfo**()**

**{**

var infoTask = GetLeisureHours**()**;

// You can do other work that does not rely on integerTask before awaiting.

string ret = $"Today is {DateTime.Today:D}\n" +

"Today's hours of leisure: " +

$"{await infoTask}";

**return** ret;

**}**

**static** **async** Task**<int>** GetLeisureHours**()**

**{**

// Task.FromResult is a placeholder for actual work that returns a string.

var today = **await** Task.FromResult**<**string**>(**DateTime.Now.DayOfWeek.ToString**())**;

// The method then can process the result in some way.

**int** leisureHours;

**if** **(**today.First**()** == 'S'**)**

leisureHours = 16;

**else**

leisureHours = 5;

**return** leisureHours;

**}**

**}**

###### **Output:**

Generalized Async Return Types in C#

**Note:**The Result property that we used to retrieve the value is a blocking property. It means if we try to access the value before the async method completes its task, then the thread which is currently active is blocked until the task completes and the value is available. In most real-time applications, we need to access the value by using the “**await**” keyword instead of accessing the property directly. But the point that you need to keep in mind is that you can only use the await property from within an async method.

##### ****The async method returning Task in C#****

We need to use the Task return type when the async method is not returning any value after the execution of the method. It means the async method either does not have a return statement in it or it may contain a return statement that doesn’t return any value. Such type of async methods returns void if they run synchronously.

If we have an async method with Task return type and if we want our caller method to wait until the async method completes its execution then we need to use the await operator while calling the async method.

In the following example, the **WaitAndApologize()** async method return type is Task as it doesn’t have a return statement. We are calling this **WaitAndApologize()** async method from the **DisplayCurrentInfo()** async method. As we want to wait until the **WaitAndApologize()** method completes its execution so when calling this method from within the **DisplayCurrentInfo()**method we use the await operator.

Again from our Main() method, we are calling the **DisplayCurrentInfo()** async method and our requirement is to wait until the **DisplayCurrentInfo()** method complete its execution, so here we using the Wait() method while calling the **DisplayCurrentInfo()**method. We can not use the await operator here because the Main method is not an async method. As we know we can use the await operator only within an async method.

**public** **class** Example

**{**

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

**{**

DisplayCurrentInfo**()**.Wait**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**static** **async** Task DisplayCurrentInfo**()**

**{**

**await** WaitAndApologize**()**;

Console.WriteLine**(**$"Today is {DateTime.Now:D}"**)**;

Console.WriteLine**(**$"The current time is {DateTime.Now.TimeOfDay:t}"**)**;

Console.WriteLine**(**"The current temperature is 76 degrees."**)**;

**}**

**static** **async** Task WaitAndApologize**()**

**{**

// Task.Delay is a placeholder for actual work.

**await** Task.Delay**(**2000**)**;

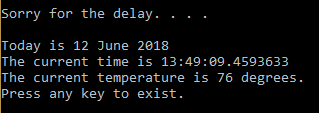
// Task.Delay delays the following line by two seconds.

Console.WriteLine**(**"\nSorry for the delay. . . .\n"**)**;

**}**

**}**

###### **Output:**



**The following code separates calling the WaitAndApologize method from awaiting the task that the method returns.**

**public** **class** Example

**{**

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

**{**

DisplayCurrentInfo**()**.Wait**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**static** **async** Task DisplayCurrentInfo**()**

**{**

Task wait = WaitAndApologize**()**;

string output = $"Today is {DateTime.Now:D}\n" +

$"The current time is {DateTime.Now.TimeOfDay:t}\n" +

$"The current temperature is 76 degrees.\n";

**await** wait;

Console.WriteLine**(**output**)**;

**}**

**static** **async** Task WaitAndApologize**()**

**{**

// Task.Delay is a placeholder for actual work.

**await** Task.Delay**(**2000**)**;

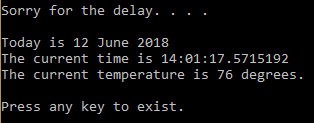
// Task.Delay delays the following line by two seconds.

Console.WriteLine**(**"\nSorry for the delay. . . .\n"**)**;

**}**

**}**

###### **Output:**



##### ****Async method returning void in C#****

We need to use the void return type in C# when the async method does not return any value. Then you may have one question in your mind **what is the difference between Task and void return types** as both are going to be used when the async method does not return any value.

The difference is that if you use the void return type then the async method cannot be awaited. That means the caller of such method (void return async method) do 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.

##### ****Example: Async method returning void in C#****

Please have a look at the below example.

**public** **class** Example

**{**

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

**{**

RunCounter**()**.Wait**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**private** **static** **async** Task RunCounter**()**

**{**

var count = new Counter**(**5**)**;

**await** count.StartCounting**(**8**)**;

**}**

**}**

**public** **class** Counter

**{**

**private** **int** threshold = 0;

**private** **int** iterations = 0;

**private** **int** ctr = 0;

**event** EventHandler**<**EventArgs**>** ThresholdReached;

**public** Counter**(int** threshold**)**

**{**

this.threshold = threshold;

ThresholdReached += thresholdReachedEvent;

**}**

**public** **async** Task**<int>** StartCounting**(int** limit**)**

**{**

iterations = 1;

**for** **(int** index = 0; index **<**= limit; index++**)**

**{**

**if** **(**ctr == threshold**)**

thresholdReachedEvent**(**this, EventArgs.Empty**)**;

ctr++;

**await** Task.Delay**(**500**)**;

**}**

**int** retval = ctr + **(**iterations - 1**)** \* threshold;

Console.WriteLine**(**$"On iteration {iterations}, reached {limit}"**)**;

**return** retval;

**}**

**async** **void** thresholdReachedEvent**(object** sender, EventArgs e**)**

**{**

Console.WriteLine**(**$"Reached {ctr}. Resetting..."**)**;

**await** Task.Delay**(**1000**)**;

ctr = 0;

iterations++;

**}**

**}**

###### **Output:**

Generalized Async Return Types in C#

I hope now you have some idea regarding the async method in C#. So, let us move to our main topic of this article i.e.**Generalized Async Return Types in C#**.

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

As of now, we have discussed the async method with return type Task, Task<T>, and void. The most important point that you need to keep in mind is that the Task is a class. We also know the reference types behave differently in C#.  In some situations, it is better to return anything rather than a Task.

The generalized async returns types in C# mean you can return a lightweight value type instead of a reference type to avoid additional memory allocations. From C# 7, there is an inbuilt value type **ValueTask <T>** which can be used instead of **Task<T>**.

.NET Framework provides the **System.Threading.Tasks.ValueTask<TResult>** as a light-weight implementation of a generalized task-returning value. To use the System.Threading.Tasks.ValueTask<TResult> type, you must add the **System.Threading.Tasks.Extensions** NuGet package to your project.

##### ****Example: Generalized Async Return Types in C#****

Let us understand Generalized Async Return Types in C# concept with an example. Please have a look at the below example. As you can see in the below example, instead of using Task<T>, now we are using ValueTask<T> which is a value type, not a reference type and because of this it will have less memory and provides better performance as compared to Task<T>.

**using** *System;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**namespace** *GeneralizedAsyncReturnTypes*

**{**

**public** **class** Example

**{**

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

**{**

Console.WriteLine**(**ShowTodaysInfo**()**.Result**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**private** **static** **async** ValueTask**<**string**>** ShowTodaysInfo**()**

**{**

var infoTask = GetLeisureHours**()**;

// You can do other work that does not rely on integerTask before awaiting.

string ret = $"Today is {DateTime.Today:D}\n" +

"Today's hours of leisure: " +

$"{await infoTask}";

**return** ret;

**}**

**static** **async** ValueTask**<int>** GetLeisureHours**()**

**{**

// Task.FromResult is a placeholder for actual work that returns a string.

var today = **await** Task.FromResult**<**string**>(**DateTime.Now.DayOfWeek.ToString**())**;

// The method then can process the result in some way.

**int** leisureHours;

**if** **(**today.First**()** == 'S'**)**

leisureHours = 16;

**else**

leisureHours = 5;

**return** leisureHours;

**}**

**}**

**}**

###### **Output:**

Generalized Async Return Types in C#

You may be thinking that we are talking about the term generalized async, but here we are using only **ValueTask<T>.** So, I would like to clarify your doubt that you can also create your own type which can be the return type of your async method. However, if you do not want to create your own type, then you can use the **ValueTask<T>** which is already available.

# Expression Bodied Members in C#

## ****Expression Bodied Members in C# with Examples****

In this article, I am going to discuss the **Expression Bodied Members in C#** with Examples. Please read our previous article where we discussed the [**Generalized Async Return Types in C#**](https://dotnettutorials.net/lesson/generalized-async-return-types/) with examples. At the end of this article, you will understand what exactly Expression Bodied Members in C# are and when and how to use this in C# with examples.

##### ****What are Expression Bodied Members in C#?****

The Expression Bodied Members in C#, allows us to provide the implementation of a member in a better readable format. You can use expression-bodied members in C# whenever the logic for any supported members such as a method or property consists of a single expression. An expression body definition has the following general syntax:

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

The Expression Bodied Members in C# was first introduced in C# 6 with only methods and properties. But with C# 7, several new members have been included in the list. The complete list of members is as shown below.

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

Let’s discuss each of these members one by one with examples.

##### ****Expression-Bodied Methods in C#:****

An expression-bodied method consists of a single expression that returns a value whose type matches the method’s return type, or, for methods that return void, that performs some operation. For example, types that override the ToString method typically include a single expression that returns the string representation of the current object.

The below example defines an Employee class that overrides the ToString method with an expression body definition. It also defines a GetFullName method that returns the full name of the employee and again It also defines a DisplayName method that displays the name to the console. Note that the return keyword is not used in the ToString expression body definition.

**public** **class** Employee

**{**

**private** string FirstName;

**private** string LastName;

**public** Employee**(**string firstName, string lastName**)**

**{**

FirstName = firstName;

LastName = lastName;

**}**

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

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

**public** **void** DisplayName**()** =**>** Console.WriteLine**(**GetFullName**())**;

**}**

**class** Program

**{**

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

**{**

Employee employee = new Employee**(**"Pranaya", "Rout"**)**;

employee.DisplayName**()**;

Console.WriteLine**(**employee**)**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

###### **OUTPUT:**

Expression Bodied Members in C#

**Note:** This removes not only the curly brackets but also the return keyword. Return is implicit with a lambda expression.

Generally, expression-bodied methods are more used than other members. They have the following characteristics.

1. Expression bodied methods can specify all the accessibility operators i.e. public, protected, internal, private, and protected internal.
2. These can be declared virtual or abstract or can even override a base class method.
3. Such methods can be static.
4. Methods can even exhibit asynchronous behavior, if they return void, Task or Task<T>.

##### ****Expression-Bodied Constructors in C#:****

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor’s arguments or initializes the instance state.

The following example defines a Location class whose constructor has a single string parameter named name. The expression body definition assigns the argument to the Name property.

**public** **class** Location

**{**

**private** string locationName;

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

**public** string Name

**{**

**get** =**>** locationName;

**set** =**>** locationName = **value**;

**}**

**}**

**class** Program

**{**

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

**{**

Location location = new Location**(**"Mumbai"**)**;

Console.WriteLine**(**location.Name**)**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

###### **OUTPUT:**

Expression Bodied Members in C#

##### ****Expression-Bodied Destructors in C#:****

An expression body definition for a destructor typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a destructor that uses an expression body definition to indicate that the destructor has been called.

**public** **class** Destroyer

**{**

**public** **override** string ToString**()** =**>** GetType**()**.Name;

~Destroyer**()** =**>** Console.WriteLine**(**$"The {ToString()} destructor is executing."**)**;

**}**

##### ****Property get****

If you choose to implement a property get accessor yourself, you can use an expression body definition for single expressions that simply return the property value. Note that the return statement isn’t used.

The following example defines the Location.Name property whose property get accessor returns the value of the private locationName field that backs the property.

**public** **class** Location

**{**

**private** string locationName;

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

**public** string Name

**{**

**get** =**>** locationName;

**set** =**>** locationName = **value**;

**}**

**}**

**class** Program

**{**

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

**{**

Location location = new Location**(**"Mumbai"**)**;

Console.WriteLine**(**location.Name**)**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

Read-only properties that use an expression body definition can be implemented without an explicit set statement. The following example defines a Location class whose read-only Name property is implemented as an expression body definition that returns the value of the private locationName field.

**public** **class** Location

**{**

**private** string locationName;

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

**public** string Name =**>** locationName;

**}**

**class** Program

**{**

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

**{**

Location location = new Location**(**"Mumbai"**)**;

Console.WriteLine**(**location.Name**)**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

The above code not only reduces the curly bracket but also writing the get accessor is not necessary. The code that’s generated from the compiler is the same.

##### ****Property set Expression Bodied Member in C#****

If you choose to implement a property set accessor yourself, you can use an expression body definition for a single-line expression that assigns a value to the field that backs the property.

The following example defines the Location.Name property whose property set statement assigns its input argument to the private locationName field that backs the property.

**public** **class** Location

**{**

**private** string locationName;

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

**public** string Name

**{**

**get** =**>** locationName;

**set** =**>** locationName = **value**;

**}**

**}**

**class** Program

**{**

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

**{**

Location location = new Location**(**"Mumbai"**)**;

Console.WriteLine**(**location.Name**)**;

location.Name = "Hyderabad";

Console.WriteLine**(**location.Name**)**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

##### ****Indexers****

Like properties, an indexer’s get and set accessors consist of expression body definitions if the get accessor consists of a single statement that returns a value or the set accessor performs a simple assignment.

The following example defines a class named Sports that includes an internal String array that contains the names of a number of sports. Both the indexer’s get and set accessors are implemented as expression body definitions.

**public** **class** Sports

**{**

**private** string**[]** types = **{**"Cricket", "Baseball", "Basketball", "Football",

"Hockey", "Soccer", "Tennis","Volleyball" **}**;

**public** string this**[int** i**]**

**{**

**get** =**>** types**[**i**]**;

**set** =**>** types**[**i**]** = **value**;

**}**

**}**

**class** Program

**{**

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

**{**

Sports sports = new Sports**()**;

Console.WriteLine**(**sports**[**0**])**;

Console.WriteLine**(**sports**[**2**])**;

Console.WriteLine**(**"Press any key to exists"**)**;

Console.ReadKey**()**;

**}**

**}**

##### ****Expression bodied Members in C#: getters  and setters****

Expression body getters and setters are also introduced in C# 7.0. They allow an expression to be used in the body of getters or setters. The example given below illustrates the same.

**class** Program

**{**

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

**{**

var obj = new ExprBodiedGettersnSetters**()**;

obj.EmpBasicSalaryList.Add**(**101, 1000**)**;

obj.EmpBasicSalaryList.Add**(**102, 1200**)**;

obj.EmpId = 101;

Console.WriteLine**(**$"The basic salary of EmpId {obj.EmpId} is: {obj.EmpBasicSalary}"**)**;

obj.EmpBasicSalary = 1500;

Console.WriteLine**(**$"The updated basic salary of EmpId {obj.EmpId} is: {obj.EmpBasicSalary}"**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

**class** ExprBodiedGettersnSetters

**{**

**public** Dictionary**<int**, **double>** EmpBasicSalaryList = new Dictionary**<int**, **double>()**;

**public** **int** EmpId **{** **get**; **set**; **}**

**public** **double** EmpBasicSalary

**{**

///Expression Bodied Getter

**get** =**>** EmpBasicSalaryList**[**EmpId**]**;

///Expression Bodied Setter

**set** =**>** EmpBasicSalaryList**[**EmpId**]** = **value**;

**}**

**}**

###### **OUTPUT:**

Expression Bodied Members in C#

##### ****Limitations of Expression Bodied Members in C#****

Although expression-bodied members in C# provide very clean syntax, they have some limitations. Let’s go through some of them and see how those can be addressed.

The Expression-bodied members in C# don’t support a block of code. It supports only one statement with an expression, which is allowed. If you need to use more than one statement then you may use the regular methods or properties.

Branching statements (if..else, switch) are not allowed however if..else behavior can be achieved by the ternary operator. For example, the statement given below can work.

**public string FullName() => (middleName != null) ? firstName + ” ” + middleName + ” ” + lastName : firstName + ” ” + lastName;**

Loop statements (i.e. for, foreach, while, and do..while are not allowed) however in some cases, it can be replaced with LINQ queries. For example, both of the following methods (HundredNumbersList and HundredNumbersListWithExprBody) return the same result.

**public IEnumerable<int> HundredNumbersList()**  
**{**  
**for (int i = 0; i < 100; i++)**  
**yield return i;**  
**}**

**public IEnumerable<int> HundredNumbersListWithExprBody() => from n in Enumerable.Range(0, 100)**  
**select n;**

##### ****Note****:

Lambda expressions can be written in short form without curly brackets when the statement consists of a single line. Lambda expressions can also be written in the long-form where curly brackets and the return statement are needed. This longer syntax is not possible with expression-bodied members in C#. If one code line is not enough, you can use the normal syntax with curly brackets, as is available since C# 1.0.

# Thrown Expression in C#

## ****Thrown Expression in C# 7 with examples****

In this article, I am going to discuss the **Thrown Expression in C#** with examples**.** Please read our previous article before proceeding to this article where we discussed the [**Expression Bodied Members in C#**](https://dotnettutorials.net/lesson/expression-bodied-members-csharp/) with examples. In C#, it is very easy to throw an exception in the middle of an expression.

##### ****Example: Thrown Expression in C#****

Let us understand Thrown Expression in C# with an example. Please have a look at the below example.

**class** Program

**{**

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

**{**

var a = Divide**(**10, 0**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**public** **static** **double** Divide**(int** x, **int** y**)**

**{**

**if** **(**y == 0**)**

**{**

**throw** new DivideByZeroException**()**;

**}**

**else**

**{**

**return** x % y;

**}**

**}**

**}**

In the above example, we are throwing an expression in the middle of the method by checking the condition. But with C# 7.0 now it is possible to throw an exception in the middle of an expression. Let’s rewrite the above program to throw an exception in the middle of the expression as shown below.

**class** Program

**{**

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

**{**

var a = Divide**(**10, 0**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**public** **static** **double** Divide**(int** x, **int** y**)**

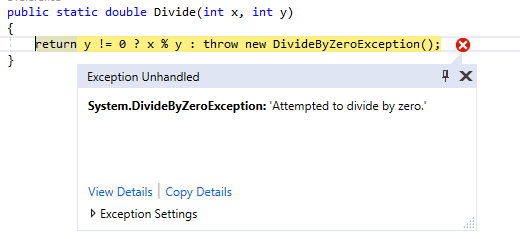
**{**

**return** y != 0 ? x % y : **throw** new DivideByZeroException**()**;

**}**

**}**

###### **OUTPUT:**



You can throw any type of exceptions in the middle of an expression like **“IndexOutOfRangeException”, “NullReferenceException”, “OutOfMemoryException”, “StackOverflowException”**and many other types of exceptions.

You can add exception throwing to expression-bodied members, null-coalescing expressions, and conditional expressions. Throw expressions are the way to tell the compiler to throw the exception under specific conditions like in expression-bodied members or inline comparisons.

##### ****Thrown Expression in C# real-time example:****

The below example uses a simple Employee class to demonstrate different situations where throw expression in C# can be used:

1. auto-property initializer statement
2. inline comparison using the operator “?”
3. expression-bodied member

**class** Employee

**{**

**public** string FullName **{** **get**; **}**

**public** Employee**(**string name**)** =**>** FullName = name ?? **throw** new ArgumentNullException**(**name**)**;

**public** string GetFirstName**()**

**{**

var parts = FullName.Split**(**' '**)**;

**return** **(**parts.Length **>** 1**)** ? parts**[**0**]** : **throw** new InvalidOperationException**(**"Method:GetFirstName, Full Name is not available"**)**;

**}**

**public** string GetLastName**()** =**>** **throw** new NotImplementedException**(**"Method GetLastName is not Implemented"**)**;

**}**

**class** Program

**{**

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

**{**

TryWithNameNull**()**;

TryGetFirstName**()**;

TryGetLastName**()**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**static** **void** TryWithNameNull**()**

**{**

**try**

**{**

new Employee**(null)**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**ex.GetType**()** + ": " + ex.Message**)**;

**}**

**}**

**static** **void** TryGetFirstName**()**

**{**

**try**

**{**

new Employee**(**"Pranaya"**)**.GetFirstName**()**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**ex.GetType**()** + ": " + ex.Message**)**;

**}**

**}**

**static** **void** TryGetLastName**()**

**{**

**try**

**{**

new Employee**(**"Pranaya"**)**.GetLastName**()**;

**}**

**catch** **(**Exception ex**)**

**{**

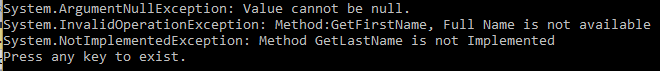
Console.WriteLine**(**ex.GetType**()** + ": " + ex.Message**)**;

**}**

**}**

**}**

###### **Output:**



# Async Main in C#

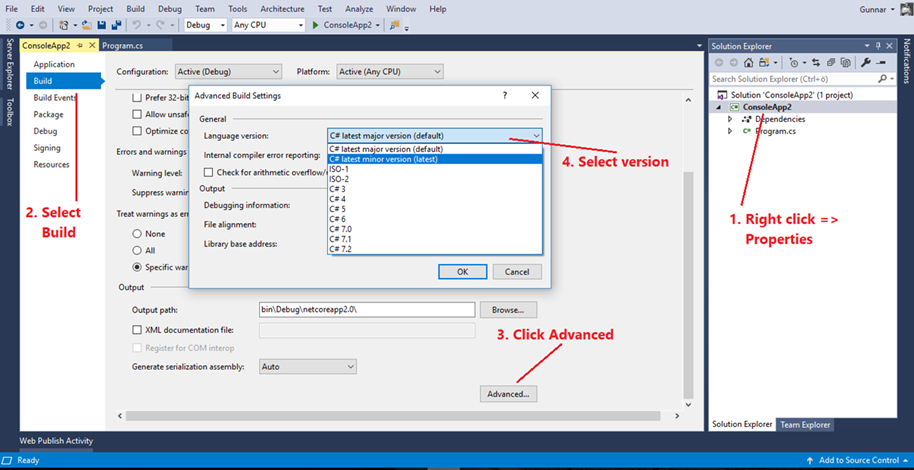
In this article, I am going to discuss the **Async Main in C#** with Examples. Please read our previous article where we discussed the [**Thrown Expression in C#**](https://dotnettutorials.net/lesson/thrown-expressions-csharp/) with some examples. From C# 7.1 now it is possible to define the Main method as Async. At the end of this article, you will understand **what exactly Async Main is in C#** and **when and how to use Async Main** with examples.

There are two types of programs that the C# language compiler can build. One is a program with an entry point so that the Operating System can load the program and execute it from the entry point. The other is the program without an entry point. Operating System cannot directly execute such types of programs. Such type of programs needs to be referenced by other programs which have an entry point.

The Application which must have an entry point includes Windows Forms App, Console App, WPF App, ASP.NET and ASP.NET Core App, and Xamarian App. On the other hand, the Applications which do not have an entry point include the Class Library.

###### **Activating C# 7.1 features**

To make Visual Studio 2017 use some other versions of C# follow the steps shown in the below image.



For those who need some more help the steps are described here:

1. Right-click on the project name and select Properties from the context menu
2. Select the Build tab from left
3. Scroll down and click the Advanced button to open advanced build settings
4. From the Language version drop-down select the C# version you are interested in

C# latest minor version (latest) is the option to select if you want the latest version of C#.

##### ****The Main Method in C#****

Like other programming languages, the C# program also starts from the **Main** method with the following properties.

1. The C# entry point method must be static,
2. The name of the method must be Main
3. The return type of this method can be either void or int.
4. It can have one parameter of a string array, containing any command-line arguments.

There are four overloaded versions that are considered as the valid signatures for the **Main** method in C# as shown below.

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

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

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

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

Before C# 7.1, when we wanted to call the async method from within the Main() method, then we need to add some code but now, C# compiler does it for us. Let’s try to understand how to call the async method from Main in C# before C# 7.1 with an example as shown below.

**class** Program

**{**

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

**{**

Console.WriteLine**(**"Before C# 7.1, To use async method"**)**;

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

Task.Delay**(**2000**)**.GetAwaiter**()**.GetResult**()**;

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

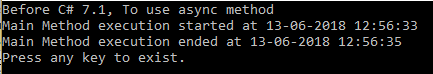
Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

###### **Output:**



##### ****Async Main in C#:****

From C# 7.1, the Main() method which is the entry point of the application can be declared as 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, now we have eight overload versions that are considered as the 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**)**;

##### ****Let us understand async main in C# with an example as shown below.****

**class** Program

**{**

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

**{**

Console.WriteLine**(**"In C# 7.1, To use async method"**)**;

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

**await** Task.Delay**(**2000**)**;

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

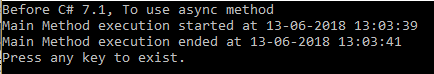
Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**}**

###### **Output:**



As you can see in the above example, the Task.Delay is adding 2 seconds delay in the program execution. Now, C# 7.1 syntax is simpler and easy to use. In the above example, we see how we could use the Task with Main. Now, let’s take another example where we will see the use of Task<int>.

Here, we will call another async method AdditionAsync from Main.

**class** Program

**{**

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

**{**

Console.Title = "async Task<int> Main";

**int** number1 = 5, number2 = 10;

Console.WriteLine**(**$"Sum of {number1} and {number2} is: {await AdditionAsync(number1, number2)}"**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**return** 0;

**}**

**private** **static** Task**<int>** AdditionAsync**(int** no1, **int** no2**)**

**{**

**return** Task.Run**(()** =**>** SUM**(**no1, no2**))**;

//Local function

**int** SUM**(int** x, **int** y**)**

**{**

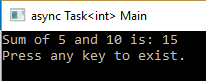
**return** x + y;

**}**

**}**

**}**

###### **Output:**



You can also see that in the above example, we have used a Local function SUM that is one of the new features of C# 7.0.