**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 before proceeding to this article where we discussed the [**Generalized Async Return Types in C#**](https://dotnettutorials.net/lesson/generalized-async-return-types/) with examples.

The Expression bodied Members in C# let you provide the implementation of a member in a better readable format. You can use an 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

**Methods**

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.

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>.

#### ****Constructors****

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 instance state.

#### ****Destructors****

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

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

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.

#### ****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.

#### ****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.

#### ****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

##### **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 statements 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 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;**

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

##### **Points to Remember while working with Local Functions in C#:**

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

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

# ****Pattern Matching in C#****

The **Pattern Matching in C#** is nothing but a mechanism which tests a value i.e. whether the value has a specific shape or not. It the value is in a specific shape then it will extract the data from the value

##### **Types of Pattern Matching:**

There are there types of Pattern Matching in C# 7, they are as follows

1. **Constant patterns:**It is of the form c where c is a constant expression, which will tests that the input value is equal to c.
2. **Type patterns:**It is of the form T x where T is a type and x is an identifier, which will tests that the input value has the type T, and if so, then extracts the values into a new variable of the type T.
3. **Var patterns:**It is of the form var x where var is the anonymous type and x is an identifier, which will always test and simply put the value of the input into a fresh variable x with the same type as the input.

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

To implement **Pattern Matching**, we have provided two language constructs such as:

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

The C# 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. The Pattern matching has the ability to extract the data from the expression.

Let us understand Pattern Matching with examples.

To implement **Pattern Matching**, we have provided two language constructs such as:

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