New features of C# 7.0

**Preface**It seems like only yesterday we got C# 6, it all happens quickly  in software development land. And now you are seeing C# 7.0. Jeez, it has  more cool features than C# 6.0 and damn sure after going through these you will also be on my side waiting for that one fine day.   
  
**For Trying C# 7.0 you need to do the following**

* [Visual Studio 15](https://www.visualstudio.com/en-us/downloads/visual-studio-next-downloads-vs.aspx)preview
* Set \_\_DEMO\_\_ and\_\_DEMO\_EXPERIMENTAL\_\_ as Conditional compilation symbol in project settings.

**Feature List in C# 7.0**

1. Local functions – code available currently in github
2. Tuple Types and literals
3. Record Types
4. Pattern matching
5. Non Nullable reference types
6. Immutable types

**Local Functions**

**Upto C# 6.0**Ability to declare methods and types in block scope as like variables. It is already possible in current version of [C#](http://www.c-sharpcorner.com/technologies/csharp-programming) using Func and Action types with anonymous methods, but they lack these features.

* Generics
* Ref and out
* Params

We can’t utilize these three features while using Func and Action.   
  
**In C# 7.0**Local functions would have the same capabilities as normal methods but they can be only accessed within the block they were declared in.

1. **public** **int** Foo(**int** someInput)
2. {
3. **int** Bar()
4. {
5. Console.WriteLine(“inner function”);
6. }
7. **return** Bar();
8. }

**Advantages over lambda expressions**

* The syntax is similar and more consistent with the methods.
* Recursion and forward references would work for the local functions but not for the lambda.
* The lambda causes one or two memory allocations (the frame object for holding the variables inside that function and a delegate); but local function causes no memory allocations.
* A local function can be generic.
* It can accept ref and out parameters.
* The direct method invocation is indeed faster than a delegate invocation.

**Tuple Types and literals  
  
Multiple return types – up to C# 6.0**In our current version of C# for returning multiple values from a method we have follow on of these methods.

1. Out parameters
2. Tuple-Types
3. Class/ Struct

The most common reason for grouping up of temporary variables is to return multiple values from a method.   
  
**Out parameters**

1. **public** **void** GetMultipleValues(**string** deptName, **out** **double** topGrossSalary, **out** **string** hrName) { ... }
3. **double** gross,
4. **string** hrName;
5. GetMultipleValues (name, **out** gross, **out** hrName);
6. Console.WriteLine($"Gross: { gross }, Hr: { hrName }");

The disadvantage of using out parameter is we can’t use it for async methods. And you have declare parameters upfront and you have specify the specific type.   
  
**Tuple-Types**  
  
Currently, C# has tuple type in order to hold multiple non related values together. We can rewrite the same method to use tuples to achieve the same functionality.

1. **public** Tuple<**double**, **string**> GetMultipleValues(**string** name) { ... }
3. var tupleSalary = GetMultipleValues (name);
4. Console.WriteLine($"Gross: { tupleSalary.Item1 }, Hr: { tupleSalary.Item2 }");

This does not have the disadvantages of out-parameters, but the resulting code is rather obscure with the resulting tuple having property names like Item1 and Item2.  
  
**Class / struct**  
  
You could also declare a new type and use that as the return type.

1. Struct TopSalaryAndHr { **public** **double** topGrossSalary; **public** **string** hrName;}
2. **public** TopSalaryAndHr GetMultipleValues(**string** name) { ... }
4. var tupleSalary = GetMultipleValues (name);
5. Console.WriteLine($"Gross: { tupleSalary.topGrossSalary }, HR: { tupleSalary.hrName }");

This has none of the disadvantages of out-parameters and the tuple type, but it’s rather verbose and it is meaningless overhead. Because, the properties inside a (class / struct) are not coming under one roof and not having any common base characteristics. For the purpose of creating temporary data structure C# has provided Tuples.  
  
All three ways mentioned above has their own disadvantages, so they want to overcome these shortcomings by introducing a miracle.   
  
**Multiple return types in C# 7.0**  
**Tuple return types:**You can specify multiple return types for a function, in much the same syntax as you do for specifying multiple input types. These are supposed to be called Tuple Types.

1. Public (**double** topGrossSalary,**string** hrName) GetMultipleValues(**string** name) {……….. }

The syntax (double topGrossSalary,string hrName) indicates an anonymous struct type with public fields of the given names and types. Note that this is different from some notions of tuple, where the members are not given names but only positions (i.e Tuple<double, string>). This is a common complaint, though, essentially degrading the consumption scenario to that of System.Tuple above. For full usefulness, tuples members need to have names. This is also fully compatible with async.

1. **public** async Task<(**double** topGrossSalary, **string** hrName)> GetMultipleValues Async(**string** name) { ... }
3. var t = await GetMultipleValues (myValues);
4. Console.WriteLine($"Sum: {t.sum}, count: {t.count}");

**Tuple literals**  
Tuple values could be created as,

1. var t = **new** (**int** sum, **int** count) { sum = 0, count = 0 };

Creating a tuple value of a known target type, should enable leaving out the member names.

1. **public** (**int** sum, **int** count) Tally(IEnumerable<**int**> values)
2. {
3. var s = 0; var c = 0;
4. **foreach** (var value **in** values) { s += value; c++; }
5. **return** (s, c); // target typed to (int sum, int count)
6. }

In the above example we have created a method with return type of tuple type with known target type and constructed a tuple type inline using the previously declared individual variables (i.e. return(s, c)).  
  
Using named arguments as a syntax analogy it may also be possible to give the names of the tuple fields directly in the literal:

1. **public** (**int** sum, **int** count) Tally(IEnumerable<**int**> values)
2. {
3. var res = (sum: 0, count: 0); // infer tuple type from names and values
4. **foreach** (var value **in** values) { res.sum += value; res.count++; }
5. **return** res;
6. }

**Tuple deconstruction**We don’t need to the tuple object as a whole because it doesn’t represent a particular entity or a thing, so the consumer of a tuple type doesn’t want to access the tuple itself, and instead he can access the internal values of the tuple.   
  
Instead of accessing the tuple properties as in the example of Tuple Return Types, you can also de-structure the tuple immediately:

1. (var sal, var hrName) = GetMultipleValues("some address");
2. Console.WriteLine($"Salary: { sal }, Hr Name: {hrName}");

**Pattern Matching  
  
Is Expression**The “is” operator can be used to test an expression against a pattern. As part of the pattern-matching feature repurposing the “is” operator to take a pattern on the right-hand-side.

1. relational\_expression : relational\_expression 'is' pattern;

It is a compile-time error if the relational\_expression to the left of the “is” token does not designate a value or does not have a type. Every identifier of the pattern introduces a new local variable that is definitely assigned after the “is” operator is true (i.e. definitely assigned when true).  
  
**Pattern**Patterns are used in the is operator and in a switch\_statement to express the shape of data against which incoming data is to be compared.  
  
There are many areas where we can use patterns in c#. You can do pattern matching on any data type, even your own, whereas if/else you always need primitives to match. Pattern matching can extract values from your expression.  
  
**For ex:**I am having handful of types

1. **class** Person(**string** Name);
3. **class** Student(**string** Name, **double** Gpa) : Person(Name);
5. **class** Teacher(**string** Name, **string** Subject) : Person(Name);
7. //This sample uses the latest c# feature record type to create objects

I wanted to perform some operations by type specific.

1. **static** **string** PrintedForm(Person p)
2. {
3. Student s;
4. Teacher t;
5. **if** ((s = p **as** Student) != **null** && s.Gpa > 3.5)
6. {
7. **return** $"Honor Student {s.Name} ({s.Gpa})";
8. }
9. **else** **if** (s != **null**)
10. {
11. **return** $"Student {s.Name} ({s.Gpa})";
12. }
13. **else** **if** ((t = p **as** Teacher) != **null**)
14. {
15. **return** $"Teacher {t.Name} of {t.Subject}";
16. }
17. **else**
18. {
19. **return** $"Person {p.Name}";
20. }
21. }

Below is the client application which consumes the printed form function,

1. **static** **void** Main(**string**[] args)
2. {
3. Person[] oa = {
4. **new** Student("Einstein", 4.0),
5. **new** Student("Elvis", 3.0),
6. **new** Student("Poindexter", 3.2),
7. **new** Teacher("Feynmann", "Physics"),
8. **new** Person("Anders"),
9. };
10. **foreach** (var o **in** oa)
11. {
12. Console.WriteLine(PrintedForm(o));
13. }
14. Console.ReadKey();
15. }

In the above sample for holding the objects, I need to create temporary variables s and t. So if I have n number of objects I need to create N temporary variables with distinct names unnecessarily, which make my code more verbose. And also the temporary variables are only need for a particular code block but it is having scope throughout the function. Note the need to declare variables s and t ahead of time even though it is used in one of the code blocks.   
  
As part of the pattern-matching feature we are repurposing the “is” operator to take a pattern on the right-hand-side. And one kind of pattern is a variable declaration. That allows us to simplify the code like this,

1. **static** **string** PrintedForm(Person p)
2. {
3. **if** (p **is** Student s && s.Gpa > 3.5) //!
4. {
5. **return** $"Honor Student {s.Name} ({s.Gpa})";
6. }
7. **else** **if** (p **is** Student s)
8. {
9. **return** $"Student {s.Name} ({s.Gpa})";
10. }
11. **else** **if** (p **is** Teacher t)
12. {
13. **return** $"Teacher {t.Name} of {t.Subject}";
14. }
15. **else**
16. {
17. **return** $"Person {p.Name}";
18. }
19. }

Now you can see that the temporary variables s and t are only declared and scoped just to the place they need to be. The switch statement is also repurposed like the case branches can also have patterns instead of just constants.

1. **static** **string** PrintedForm(Person p)
2. {
3. **switch** (p) //!
4. {
5. **case** Student s when s.Gpa > 3.5 :
6. **return** $"Honor Student {s.Name} ({s.Gpa})";
7. **case** Student s :
8. **return** $"Student {s.Name} ({s.Gpa})";
9. **case** Teacher t :
10. **return** $"Teacher {t.Name} of {t.Subject}";
11. **default** :
12. **return** $"Person {p.Name}";
13. }
14. }

You can see in the above example that the case statements with patterns. Please note the new when key word in switch statement.  
  
**Record Types**Record Types is concept used for creating a type with only properties. By using that we can embed the constructor declaration with the class declaration.   
  
**For ex:**

1. Class Student(**string** Name, **int** Age);

This simple statement would automatically generate the following code inbuilt.

1. Class Student
2. {
3. **string** \_name;
4. **int** \_age;
6. **public** Person(**string** Name, **int** Age)
7. {
8. **this**.Name = Name;
9. **this**.Age = Age;
10. }
11. **public** **string** Name {**get**{ **return** **this**.\_name;}}
12. **public** **int** Age {**get**{ **return** **this**.\_age;}}
13. }

* Read-only properties, thus creating it as immutable type.
* The class will automatically implement Equality implementations like (such as GetHashCode, Equals, operator ==, operator != and so forth).
* A default implementation of ToString() method.

**Non-Nullable reference types:**Non- nullable reference option will let you create a reference type that is guaranteed not to be null. NullReference expections are too common in a project. Often we developers forgot to check a reference type for null before accessing the properties of it, thus paving way to problems.  
  
Either we forget check for it making our code vulnerable to runtime exceptions or we will check for it which makes our code more verbose.  
  
Instead of using the “?” for identifying the nullable value type we are going to use “!”.The currently proposed syntax is as follows:

1. **int** a; //non-nullable value type
2. **int**? b; //nullable value type
3. **string**! c; //non-nullable reference type
4. **string** d; //nullable reference type
6. MyClass a; // Nullable reference type
7. MyClass! b; // Non-nullable reference type
9. a = **null**; // OK, this is nullable
10. b = **null**; // Error, b is non-nullable
11. b = a; // Error, a might be null, b can't be null
13. WriteLine(b.ToString()); // OK, can't be null
14. WriteLine(a.ToString()); // Warning! Could be null!
16. **if** (a != **null**) { WriteLine(a.ToString); } // OK, you checked
17. WriteLine(a!.Length); // Ok, if you say so
19. It would be quite problematic **using** the same syntax **for** generic types and collections. For example
20. // The Dictionary is non-nullable but string, List and MyClass aren't
21. Dictionary<**string**, List<MyClass>>! myDict;
23. // Proper way to declare all types as non-nullable
24. Dictionary<**string**!, List<MyClass!>!>! myDict;

For specifying all the arguments in a collection is non- nullable, there is a shortcut syntax has been proposed,

1. // Typing ! in front of the type arguments makes all types non-nullable
2. Dictionary!<**string**, List<MyClass>> myDict;

**Immutable Types**An immutable object is an object whose state cannot be changed after its creation, which means Immutable objects are objects which once loaded cannot be changed / modified by any way external or internal.  
  
Immutable objects offer few benefits,

* Inherently thread-safe.
* Easier to parallelize.
* Makes it easier to use and reason about code.
* Reference to immutable objects can be cached, as they won’t change.

Currently it is also possible to create immutable classes. Create a class with properties only with get and read-only and constant private variables.

1. Public **class** Point
2. {
3. **public** Point(**int** x, **int** y)
4. {
5. x = x;
6. Y = y;
7. }
8. **public** **int** X { **get**; }
9. **public** **int** Y { **get**; }
10. }

Code in the above example is definitely an immutable class, but the intent of the class is not clearly stated as immutable. Because, in future any one can add setters to any of the properties thus making it as mutable.   
  
The proposed syntax for creating an immutable class will force the developer to strictly adhere the rules hence will make the class an immutable. Below is the proposed syntax,

1. **public** immutable **class** Point
2. {
3. **public** Point(**int** x, **int** y)
4. {
5. x = x;
6. Y = y;
7. }
9. **public** **int** X { **get**; }
10. **public** **int** Y { **get**; }
11. }