New Features in C# 7.0

Here is a description of all the new language features in C# 7.0, which came out last Tuesday as part of the Visual Studio 2017 release.

Out variables

In older versions of C#, using out parameters isn't as fluid as we'd like. Before you can call a method with out parameters you first have to declare variables to pass to it. Since you typically aren't initializing these variables (they are going to be overwritten by the method after all), you also cannot use var to declare them, but need to specify the full type:

```
public void PrintCoordinates(Point p)
{
    int x, y; // have to "predeclare"
    p.GetCoordinates(out x, out y);
    WriteLine($"({x}, {y})");
}
```


In C# 7.0 we have added ***out variables***; the ability to declare a variable right at the point where it is passed as an out argument:

```
public void PrintCoordinates(Point p)
{
    p.GetCoordinates(out int x, out int y);
    WriteLine($"({x}, {y})");
}
```


Note that the variables are in scope in the enclosing block, so that the subsequent line can use them. Many kinds of statements do not establish their own scope, so out variables declared in them are often introduced into the enclosing scope.

Since the out variables are declared directly as arguments to out parameters, the compiler can usually tell what their type should be (unless there are conflicting overloads), so it is fine to use var instead of a type to declare them:

```
p.GetCoordinates(out var x, out var y);
```


A common use of out parameters is the Try... pattern, where a boolean return value indicates success, and out parameters carry the results obtained:

```
public void PrintStars(string s)
{
    if (int.TryParse(s, out var i)) { WriteLine(new string('*', i)); }
    else { WriteLine("Cloudy - no stars tonight!"); }
}
```


We allow "discards" as out parameters as well, in the form of a _, to let you ignore out parameters you don't care about:

```
p.GetCoordinates(out var x, out _); // I only care about x
```


Pattern matching

C# 7.0 introduces the notion of ***patterns***, which, abstractly speaking, are syntactic elements that can *test* that a value has a certain "shape", and *extract* information from the value when it does.

Examples of patterns in C# 7.0 are:

- *Constant patterns* of the form c (where c is a constant expression in C#), which test that the input is equal to c
- *Type patterns* of the form T x (where T is a type and x is an identifier), which test that the input has type T, and if so, extracts the value of the input into a fresh variable x of type T
- *Var patterns* of the form var x (where x is an identifier), which always match, and simply put the value of the input into a fresh variable x with the same type as the input.

This is just the beginning – patterns are a new kind of language element in C#, and we expect to add more of them to C# in the future.

In C# 7.0 we are enhancing two existing language constructs with patterns:

- is expressions can now have a pattern on the right hand side, instead of just a type
- case clauses in switch statements can now match on patterns, not just constant values

In future versions of C# we are likely to add more places where patterns can be used.

Is-expressions with patterns

Here is an example of using is expressions with constant patterns and type patterns:

```
public void PrintStars(object o)
{
    if (o is null) return;     // constant pattern "null"
    if (!(o is int i)) return; // type pattern "int i"
    WriteLine(new string('*', i));
}
```


As you can see, the ***pattern variables*** – the variables introduced by a pattern – are similar to the out variables described earlier, in that they can be declared in the middle of an expression, and can be used within the nearest surrounding scope. Also like out variables, pattern variables are mutable. We often refer to out variables and pattern variables jointly as "expression variables".

Patterns and Try-methods often go well together:

```
if (o is int i || (o is string s && int.TryParse(s, out i)) { /* use i */ }
```


Switch statements with patterns

We're generalizing the switch statement so that:

- You can switch on any type (not just primitive types)
- Patterns can be used in case clauses
- Case clauses can have additional conditions on them

Here's a simple example:

```
switch(shape)
{
    case Circle c:
        WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle s when (s.Length == s.Height):
        WriteLine($"{s.Length} x {s.Height} square");
        break;
    case Rectangle r:
        WriteLine($"{r.Length} x {r.Height} rectangle");
        break;
    default:
        WriteLine("<unknown shape>");
        break;
    case null:
        throw new ArgumentNullException(nameof(shape));
}
```


There are several things to note about this newly extended switch statement:

- *The order of case clauses now matters:* Just like catch clauses, the case clauses are no longer necessarily disjoint, and the first one that matches gets picked. It's therefore important that the square case comes before the rectangle case above. Also, just like with catch clauses, the compiler will help you by flagging obvious cases that can never be reached. Before this you couldn't ever *tell* the order of evaluation, so this is not a breaking change of behavior.
- *The default clause is always evaluated last:* Even though the null case above comes last, it will be checked before the default clause is picked. This is for compatibility with existing switch semantics. However, good practice would usually have you put the default clause at the end.
- *The null clause at the end is not unreachable:* This is because type patterns follow the example of the current is expression and do *not* match null. This ensures that null values aren't accidentally snapped up by whichever type pattern happens to come first; you have to be more explicit about how to handle them (or leave them for the default clause).

Pattern variables introduced by a case ...: label are in scope only in the corresponding switch section.

Tuples

It is common to want to return more than one value from a method. The options available in older versions of C# are less than optimal:

- *Out parameters*: Use is clunky (even with the improvements described above), and they don't work with async methods.
- System.Tuple<...> *return types*: Verbose to use and require an allocation of a tuple object.
- *Custom-built transport type for every method*: A lot of code overhead for a type whose purpose is just to temporarily group a few values.
- *Anonymous types returned through a* dynamic *return type*: High performance overhead and no static type checking.

To do better at this, C# 7.0 adds ***tuple types*** and ***tuple literals***:

```
(string, string, string) LookupName(long id) // tuple return type
{
    ... // retrieve first, middle and last from data storage
    return (first, middle, last); // tuple literal
}
```


The method now effectively returns three strings, wrapped up as elements in a tuple value.

The caller of the method will receive a tuple, and can access the elements individually:

```
var names = LookupName(id);
WriteLine($"found {names.Item1} {names.Item3}.");
```


Item1 etc. are the default names for tuple elements, and can always be used. But they aren't very descriptive, so you can optionally add better ones:

```
(string first, string middle, string last) LookupName(long id) // tuple elements have names
```


Now the recipient of that tuple have more descriptive names to work with:

```
var names = LookupName(id);
WriteLine($"found {names.first} {names.last}.");
```


You can also specify element names directly in tuple literals:

```
    return (first: first, middle: middle, last: last); // named tuple elements in a literal
```


Generally you can assign tuple types to each other regardless of the names: as long as the individual elements are assignable, tuple types convert freely to other tuple types.

Tuples are value types, and their elements are simply public, mutable fields. They have value equality, meaning that two tuples are equal (and have the same hash code) if all their elements are pairwise equal (and have the same hash code).

This makes tuples useful for many other situations beyond multiple return values. For instance, if you need a dictionary with multiple keys, use a tuple as your key and everything works out right. If you need a list with multiple values at each position, use a tuple, and searching the list etc. will work correctly.

Tuples rely on a family of underlying generic struct types called ValueTuple<...>. If you target a Framework that doesn't yet include those types, you can instead pick them up from NuGet:

- Right-click the project in the Solution Explorer and select "Manage NuGet Packages..."
- Select the "Browse" tab and select "nuget.org" as the "Package source"
- Search for "System.ValueTuple" and install it.

Deconstruction

Another way to consume tuples is to *deconstruct* them. A ***deconstructing declaration*** is a syntax for splitting a tuple (or other value) into its parts and assigning those parts individually to fresh variables:

```
(string first, string middle, string last) = LookupName(id1); // deconstructing declaration
WriteLine($"found {first} {last}.");
```


In a deconstructing declaration you can use var for the individual variables declared:

```
(var first, var middle, var last) = LookupName(id1); // var inside
```


Or even put a single var outside of the parentheses as an abbreviation:

```
var (first, middle, last) = LookupName(id1); // var outside
```


You can also deconstruct into existing variables with a ***deconstructing assignment***:

```
(first, middle, last) = LookupName(id2); // deconstructing assignment
```


Deconstruction is not just for tuples. Any type can be deconstructed, as long as it has an (instance or extension) *deconstructor method* of the form:

```
public void Deconstruct(out T1 x1, ..., out Tn xn) { ... }
```


The out parameters constitute the values that result from the deconstruction.

(Why does it use out parameters instead of returning a tuple? That is so that you can have multiple overloads for different numbers of values).

```
class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) { X = x; Y = y; }
    public void Deconstruct(out int x, out int y) { x = X; y = Y; }
}

(var myX, var myY) = GetPoint(); // calls Deconstruct(out myX, out myY);
```


It will be a common pattern to have constructors and deconstructors be "symmetric" in this way.

Just as for out variables, we allow "discards" in deconstruction, for things that you don't care about:

```
(var myX, _) = GetPoint(); // I only care about myX
```


Local functions

Sometimes a helper function only makes sense inside of a single method that uses it. You can now declare such functions inside other function bodies as a ***local function***:

```
public int Fibonacci(int x)
{
    if (x < 0) throw new ArgumentException("Less negativity please!", nameof(x));
    return Fib(x).current;

    (int current, int previous) Fib(int i)
    {
        if (i == 0) return (1, 0);
        var (p, pp) = Fib(i - 1);
        return (p + pp, p);
    }
}
```


Parameters and local variables from the enclosing scope are available inside of a local function, just as they are in lambda expressions.

As an example, methods implemented as iterators commonly need a non-iterator wrapper method for eagerly checking the arguments at the time of the call. (The iterator itself doesn't start running until MoveNext is called). Local functions are perfect for this scenario:

```
public IEnumerable<T> Filter<T>(IEnumerable<T> source, Func<T, bool> filter)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (filter == null) throw new ArgumentNullException(nameof(filter));

    return Iterator();

    IEnumerable<T> Iterator()
    {
        foreach (var element in source)
        {
            if (filter(element)) { yield return element; }
        }
    }
}
```


If Iterator had been a private method next to Filter, it would have been available for other members to accidentally use directly (without argument checking). Also, it would have needed to take all the same arguments as Filter instead of having them just be in scope.

Literal improvements

C# 7.0 allows _ to occur as a ***digit separator*** inside number literals:

```
var d = 123_456;
var x = 0xAB_CD_EF;
```


You can put them wherever you want between digits, to improve readability. They have no effect on the value.

Also, C# 7.0 introduces ***binary literals***, so that you can specify bit patterns directly instead of having to know hexadecimal notation by heart.

```
var b = 0b1010_1011_1100_1101_1110_1111;
```


Ref returns and locals

Just like you can pass things by reference (with the ref modifier) in C#, you can now *return* them by reference, and also store them by reference in local variables.

```
public ref int Find(int number, int[] numbers)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == number)
        {
            return ref numbers[i]; // return the storage location, not the value
        }
    }
    throw new IndexOutOfRangeException($"{nameof(number)} not found");
}

int[] array = { 1, 15, -39, 0, 7, 14, -12 };
ref int place = ref Find(7, array); // aliases 7's place in the array
place = 9; // replaces 7 with 9 in the array
WriteLine(array[4]); // prints 9
```


This is useful for passing around placeholders into big data structures. For instance, a game might hold its data in a big preallocated array of structs (to avoid garbage collection pauses). Methods can now return a reference directly to such a struct, through which the caller can read and modify it.

There are some restrictions to ensure that this is safe:

- You can only return refs that are "safe to return": Ones that were passed to you, and ones that point into fields in objects.
- Ref locals are initialized to a certain storage location, and cannot be mutated to point to another.

Generalized async return types

Up until now, async methods in C# must either return void, Task or Task<T>. C# 7.0 allows other types to be defined in such a way that they can be returned from an async method.

For instance we now have a ValueTask<T> struct type. It is built to prevent the allocation of a Task<T> object in cases where the result of the async operation is already available at the time of awaiting. For many async scenarios where buffering is involved for example, this can drastically reduce the number of allocations and lead to significant performance gains.

There are many other ways that you can imagine custom "task-like" types being useful. It won't be straightforward to create them correctly, so we don't expect most people to roll their own, but it is likely that they will start to show up in frameworks and APIs, and callers can then just return and await them the way they do Tasks today.

More expression bodied members

Expression bodied methods, properties etc. are a big hit in C# 6.0, but we didn't allow them in all kinds of members. C# 7.0 adds accessors, constructors and finalizers to the list of things that can have expression bodies:

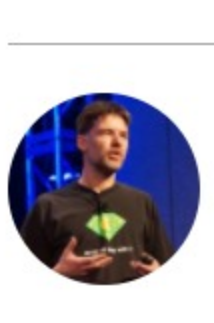
```
class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();
    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors
    ~Person() => names.TryRemove(id, out _);              // finalizers
    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```


This is an example of a feature that was contributed by the community, not the Microsoft C# compiler team. Yay, open source!

Throw expressions

It is easy to throw an exception in the middle of an expression: just call a method that does it for you! But in C# 7.0 we are directly allowing throw as an expression in certain places:

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));
    public string GetFirstName()
    {
        var parts = Name.Split(" ");
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");
    }
    public string GetLastName() => throw new NotImplementedException();
}
```

Mads Torgersen
C# Lead Designer, .NET Team
Follow