What is new in C# 7

C# 7 adds a number of new features to the C# language:

- `out` variables:
  - You can declare `out` values inline as arguments to the method where they are used.
- Tuples
  - You can create lightweight, unnamed types that contain multiple public fields. Compilers and IDE tools understand the semantics of these types.
- Pattern Matching
  - You can create branching logic based on arbitrary types and values of the members of those types.
- `ref` locals and returns
  - Method arguments and local variables can be references to other storage.
- Local Functions
  - You can nest functions inside other functions to limit their scope and visibility.
- More expression-bodied members
  - The list of members that can be authored using expressions has grown.
- `throw` Expressions
  - You can throw exceptions in code constructs that previously were not allowed because `throw` was a statement.
- Generalized async return types
  - Methods declared with the `async` modifier can return other types in addition to `Task` and `Task<T>`.
- Numeric literal syntax improvements
  - New tokens improve readability for numeric constants.

The remainder of this topic discusses each of the features. For each feature, you'll learn the reasoning behind it. You'll learn the syntax. You'll see some sample scenarios where using the new feature will make you more productive as a developer.

`out` variables

The existing syntax that supports `out` parameters has been improved in this version.

Previously, you would need to separate the declaration of the out variable and its initialization into two different statements:

C# Copy

```
int numericResult;
if (int.TryParse(input, out numericResult))
    WriteLine(numericResult);
else
    WriteLine("Could not parse input");
```

You can now declare `out` variables in the argument list of a method call, rather than writing a separate declaration statement:

C# Copy

```
if (int.TryParse(input, out int result))
    WriteLine(result);
else
    WriteLine("Could not parse input");
```

You may want to specify the type of the `out` variable for clarity, as shown above. However, the language does support using an implicitly typed local variable:

C# Copy

```
if (int.TryParse(input, out var answer))
    WriteLine(answer);
else
    WriteLine("Could not parse input");
```

- The code is easier to read.
  - You declare the out variable where you use it, not on another line above.
- No need to assign an initial value.
  - By declaring the `out` variable where it is used in a method call, you can't accidentally use it before it is assigned.

The most common use for this feature will be the `Try` pattern. In this pattern, a method returns a `bool` indicating success or failure and an `out` variable that provides the result if the method succeeds.

When using the `out` variable declaration, the declared variable "leaks" into the outer scope of the if statement. This allows you to use the variable afterwards:

C# Copy

```
if (!int.TryParse(input, out int result))
{
    return null;
}

return result;
```

Tuples

C# provides a rich syntax for classes and structs that is used to explain your design intent. But sometimes that rich syntax requires extra work with minimal benefit. You may often write methods that need a simple structure containing more than one data element. To support these scenarios *tuples* were added to C#. Tuples are lightweight data structures that contain multiple fields to represent the data members. The fields are not validated, and you cannot define your own methods

Note

Tuples were available before C# 7 as an API, but had many limitations. Most importantly, the members of these tuples were named `Item1`, `Item2` and so on. The language support enables semantic names for the fields of a Tuple.

You can create a tuple by assigning each member to a value:

C# Copy

```
var letters = ("a", "b");
```

That assignment creates a tuple whose members are `Item1` and `Item2`, following the existing Tuple syntax. You can modify that assignment to create a tuple that provides semantic names to each of the members of the tuple:

C# Copy

```
(string Alpha, string Beta) namedLetters = ("a", "b");
```

Note

The new tuples features require the `System.ValueTuple` type. For Visual Studio 2017, you must add the NuGet package System.ValueTuple, available on the NuGet Gallery.

The `namedLetters` tuple contains fields referred to as `Alpha` and `Beta`. In a tuple assignment, you can also specify the names of the fields on the right-hand side of the assignment:

C# Copy

```
var alphabetStart = (Alpha: "a", Beta: "b");
```

The language allows you to specify names for the fields on both the left and right-hand side of the assignment:

C# Copy

```
(string First, string Second) firstLetters = (Alpha: "a", Beta: "b");
```

The line above generates a warning, `CS8123`, telling you that the names on the right side of the assignment, `Alpha` and `Beta` are ignored because they conflict with the names on the left side, `First` and `Second`.

The examples above show the basic syntax to declare tuples. Tuples are most useful as return types for `private` and `internal` methods. Tuples provide a simple syntax for those methods to return multiple discrete values: You save the work of authoring a `class` or a `struct` that defines the type returned. There is no need for creating a new type.

Creating a tuple is more efficient and more productive. It is a simpler, lightweight syntax to define a data structure that carries more than one value. The example method below returns the minimum and maximum values found in a sequence of integers:

C# Copy

```
private static (int Max, int Min) Range(IEnumerable<int> numbers)
{
    int min = int.MaxValue;
    int max = int.MinValue;
    foreach(var n in numbers)
    {
        min = (n < min) ? n : min;
        max = (n > max) ? n : max;
    }
    return (max, min);
}
```

Using tuples in this way offers several advantages:

- You save the work of authoring a `class` or a `struct` that defines the type returned.
- You do not need to create new type.
- The language enhancements removes the need to call the Create<T1>(T1) methods.

The declaration for the method provides the names for the fields of the tuple that is returned. When you call the method, the return value is a tuple whose fields are `Max` and `Min`:

C# Copy

```
var range = Range(numbers);
```

There may be times when you want to unpackage the members of a tuple that were returned from a method. You can do that by declaring separate variables for each of the values in the tuple. This is called *deconstructing* the tuple:

C# Copy

```
(int max, int min) = Range(numbers);
```

You can also provide a similar deconstruction for any type in .NET. This is done by writing a `Deconstruct` method as a member of the class. That `Deconstruct` method provides a set of `out` arguments for each of the properties you want to extract. Consider this `Point` class that provides a deconstructor method that extracts the `X` and `Y` coordinates:

C# Copy

```
public class Point
{
    public Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y)
    {
        x = this.X;
        y = this.Y;
    }
```

```
}
```

You can extract the individual fields by assigning a tuple to a `Point`:

C# Copy

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

You are not bound by the names defined in the `Deconstruct` method. You can rename the extract variables as part of the assignment:

C# Copy

```
(double horizontalDistance, double verticalDistance) = p;
```

You can learn more in depth about tuples in the tuples topic.

Pattern matching

Pattern matching is a feature that allows you to implement method dispatch on properties other than the type of an object. You're probably already familiar with method dispatch based on the type of an object. In Object Oriented programming, virtual and override methods provide language syntax to implement method dispatching based on an object's type. Base and Derived classes provide different implementations. Pattern matching expressions extend this concept so that you can easily implement similar dispatch patterns for types and data elements that are not related through an inheritance hierarchy.

Pattern matching supports `is` expressions and `switch` expressions. Each enables inspecting an object and its properties to determine if that object satisfies the sought pattern. You use the `when` keyword to specify additional rules to the pattern.

`is` expression

The `is` pattern expression extends the familiar `is` operator to query an object beyond its type.

Let's start with a simple scenario. We'll add capabilities to this scenario that demonstrate how pattern matching expressions make algorithms that work with unrelated types easy. We'll start with a method that computes the sum of a number of die rolls:

C# Copy

```
public static int DiceSum(IEnumerable<int> values)
{
    return values.Sum();
}
```

You might quickly find that you need to find the sum of die rolls where some of the rolls are made with more than one die. Part of the input sequence may be multiple results instead of a single number:

C# Copy

```
public static int DiceSum2(IEnumerable<object> values)
{
    var sum = 0;
    foreach(var item in values)
    {
        if (item is int val)
            sum += val;
        else if (item is IEnumerable<object> subList)
            sum += DiceSum2(subList);
    }
    return sum;
}
```

The `is` pattern expression works quite well in this scenario. As part of checking the type, you write a variable initialization. This creates a new variable of the validated runtime type.

As you keep extending these scenarios, you may find that you build more `if` and `else` `if` statements. Once that becomes unwieldy, you'll likely want to switch to `switch` pattern expressions.

`switch` statement updates

The *match expression* has a familiar syntax, based on the `switch` statement already part of the C# language. Let's translate the existing code to use a match expression before adding new cases:

C# Copy

```
public static int DiceSum3(IEnumerable<object> values)
{
    var sum = 0;
    foreach (var item in values)
    {
        switch (item)
        {
            case int val:
                sum += val;
                break;
            case IEnumerable<object> subList:
                sum += DiceSum3(subList);
                break;
        }
    }
    return sum;
}
```

The match expressions have a slightly different syntax than the `is` expressions, where you declare the type and variable at the beginning of the `case` expression.

The match expressions also support constants. This can save time by factoring out simple cases:

C# Copy

```
public static int DiceSum4(IEnumerable<object> values)
{
    var sum = 0;
    foreach (var item in values)
    {
        switch (item)
        {
            case 0:
                break;
            case int val:
                sum += val;
                break;
            case IEnumerable<object> subList when subList.Any():
                sum += DiceSum4(subList);
                break;
            case IEnumerable<object> subList:
                break;
            case null:
                break;
            default:
                throw new InvalidOperationException("unknown item type");
        }
    }
    return sum;
```

```
}
```

The code above adds cases for `0` as a special case of `int`, and `null` as a special case when there is no input. This demonstrates one important new feature in switch pattern expressions: the order of the `case` expressions now matters. The `0` case must appear before the general `int` case. Otherwise, the first pattern to match would be the `int` case, even when the value is `0`. If you accidentally order match expressions such that a later case has already been handled, the compiler will flag that and generate an error.

This same behavior enables the special case for an empty input sequence. You can see that the case for an `IEnumerable` item that has elements must appear before the general `IEnumerable` case.

This version has also added a `default` case. The `default` case is always evaluated last, regardless of the order it appears in the source. For that reason, convention is to put the `default` case last.

Finally, let's add one last `case` for a new style of die. Some games use percentile dice to represent larger ranges of numbers.

Note

Two 10-sided percentile dice can represent every number from 0 through 99. One die has sides labelled `00`, `10`, `20`, ... `90`. The other die has sides labeled `0`, `1`, `2`, ... `9`. Add the two die values together and you can get every number from 0 through 99.

To add this kind of die to your collection, first define a type to represent the percentile die:

C# Copy

```
public struct PercentileDie
{
    public int Value { get; }
    public int Multiplier { get; }

    public PercentileDie(int multiplier, int value)
    {
        this.Value = value;
        this.Multiplier = multiplier;
    }
}
```

Then, add a `case` match expression for the new type:

C# Copy

```
public static int DiceSum5(IEnumerable<object> values)
{
    var sum = 0;
    foreach (var item in values)
    {
        switch (item)
        {
            case 0:
                break;
            case int val:
                sum += val;
                break;
            case PercentileDie die:
                sum += die.Multiplier * die.Value;
                break;
            case IEnumerable<object> subList when subList.Any():
                sum += DiceSum5(subList);
                break;
            case IEnumerable<object> subList:
                break;
            case null:
                break;
            default:
                throw new InvalidOperationException("unknown item type");
        }
    }
    return sum;
}
```

The new syntax for pattern matching expressions makes it easier to create dispatch algorithms based on an object's type, or other properties, using a clear and concise syntax. Pattern matching expressions enable these constructs on data types that are unrelated by inheritance.

You can learn more about pattern matching in the topic dedicated to pattern matching in C#.

Ref locals and returns

This feature enables algorithms that use and return references to variables defined elsewhere. One example is working with large matrices, and finding a single location with certain characteristics. One method would return the two indices for a single location in the matrix:

C# Copy

```
public static (int i, int j) Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return (i, j);
    return (-1, -1); // Not found
}
```

There are many issues with this code. First of all, it's a public method that's returning a tuple. The language supports this, but user defined types (either classes or structs) are preferred for public APIs.

Second, this method is returning the indices to the item in the matrix. That leads callers to write code that uses those indices to dereference the matrix and modify a single element:

C# Copy

```
var indices = MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(indices);
matrix[indices.i, indices.j] = 24;
```

You'd rather write a method that returns a *reference* to the element of the matrix that you want to change. You could only accomplish this by using unsafe code and returning a pointer to an `int` in previous versions.

Let's walk through a series of changes to demonstrate the ref local feature and show how to create a method that returns a reference to internal storage. Along the way, you'll learn the rules of the ref return and ref local feature that protects you from accidentally misusing it.

Start by modifying the `Find` method declaration so that it returns a `ref int` instead of a tuple. Then, modify the return statement so it returns the value stored in the matrix instead of the two indices:

C# Copy

```
// Note that this won't compile.
// Method declaration indicates ref return,
// but return statement specifies a value return.
public static ref int Find2(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
```

```
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

When you declare that a method returns a `ref` variable, you must also add the `ref` keyword to each return statement. That indicates return by reference, and helps developers reading the code later remember that the method returns by reference:

C# Copy

```
public static ref int Find3(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Now that the method returns a reference to the integer value in the matrix, you need to modify where it's called. The `var` declaration means that `valItem` is now an `int` rather than a tuple:

C# Copy

```
var valItem = MatrixSearch.Find3(matrix, (val) => val == 42);
Console.WriteLine(valItem);
valItem = 24;
Console.WriteLine(matrix[4, 2]);
```

The second `WriteLine` statement in the example above prints out the value `42`, not `24`. The variable `valItem` is an `int`, not a `ref int`. The `var` keyword enables the compiler to specify the type, but will not implicitly add the `ref` modifier. Instead, the value referred to by the `ref return` is *copied* to the variable on the left-hand side of the assignment. The variable is not a `ref` local.

In order to get the result you want, you need to add the `ref` modifier to the local variable declaration to make the variable a reference when the return value is a reference:

C# Copy

```
ref var item = ref MatrixSearch.Find3(matrix, (val) => val == 42);
Console.WriteLine(item);
```

```
item = 24;
Console.WriteLine(matrix[4, 2]);
```

Now, the second `WriteLine` statement in the example above will print out the value `24`, indicating that the storage in the matrix has been modified. The local variable has been declared with the `ref` modifier, and it will take a `ref` return. You must initialize a `ref` variable when it is declared, you cannot split the declaration and the initialization.

The C# language has two other rules that protect you from misusing the `ref` locals and returns:

- You cannot assign a value to a `ref` variable.
  - That disallows statements like `ref int i = sequence.Count();`
- You cannot return a `ref` to a variable whose lifetime does not extend beyond the execution of the method.
  - That means you cannot return a reference to a local variable, or similar scope.

These rules ensure that you cannot accidentally mix value variables and reference variables. They also ensure that you cannot have a reference variable refer to storage that is a candidate for garbage collection.

The addition of ref locals and ref returns enable algorithms that are more efficient by avoiding copying values, or performing dereferencing operations multiple times.

Local functions

Many designs for classes include methods that are called from only one location. These additional private methods keep each method small and focused. However, they can make it harder to understand a class when reading it the first time. These methods must be understood outside of the context of the single calling location.

For those designs, *local functions* enable you to declare methods inside the context of another method. This makes it easier for readers of the class to see that the local method is only called from the context in which is it declared.

There are two very common use cases for local functions: public iterator methods and public async methods. Both types of methods generate code that reports errors later than programmers might expect. In the case of iterator methods, any exceptions are

observed only when calling code that enumerates the returned sequence. In the case of async methods, any exceptions are only observed when the returned `Task` is awaited.

Let's start with an iterator method:

C# Copy

```
public static IEnumerable<char> AlphabetSubset(char start, char end)
{
    if ((start < 'a') || (start > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message:
"start must be a letter");
    if ((end < 'a') || (end > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end
must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than
{nameof(start)}");
    for (var c = start; c < end; c++)
        yield return c;
}
```

Examine the code below that calls the iterator method incorrectly:

C# Copy

```
var resultSet = Iterator.AlphabetSubset('f', 'a');
Console.WriteLine("iterator created");
foreach (var thing in resultSet)
    Console.Write($"{thing}, ");
```

The exception is thrown when `resultSet` is iterated, not when `resultSet` is created. In this contained example, most developers could quickly diagnose the problem. However, in larger codebases, the code that creates an iterator often isn't as close to the code that enumerates the result. You can refactor the code so that the public method validates all arguments, and a private method generates the enumeration:

C# Copy

```
public static IEnumerable<char> AlphabetSubset2(char start, char end)
{
    if ((start < 'a') || (start > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message:
"start must be a letter");
    if ((end < 'a') || (end > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end
must be a letter");

    if (end <= start)
```

```
        throw new ArgumentException($"{nameof(end)} must be greater than
{nameof(start)}");
    return alphabetSubsetImplementation(start, end);
}

private static IEnumerable<char> alphabetSubsetImplementation(char start, char end)
{
    for (var c = start; c < end; c++)
        yield return c;
}
```

This refactored version will throw exceptions immediately because the public method is not an iterator method; only the private method uses the `yield return` syntax. However, there are potential problems with this refactoring. The private method should only be called from the public interface method, because otherwise all argument validation is skipped. Readers of the class must discover this fact by reading the entire class and searching for any other references to the `alphabetSubsetImplementation` method.

You can make that design intent more clear by declaring the `alphabetSubsetImplementation` as a local function inside the public API method:

C# Copy

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if ((start < 'a') || (start > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message:
"start must be a letter");
    if ((end < 'a') || (end > 'z'))
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end
must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than
{nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

The version above makes it clear that the local method is referenced only in the context of the outer method. The rules for local functions also ensure that a developer can't

accidentally call the local function from another location in the class and bypass the argument validation.

The same technique can be employed with `async` methods to ensure that exceptions arising from argument validation are thrown before the asynchronous work begins:

C# Copy

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName:
nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The
index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName:
nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Note

Some of the designs that are supported by local functions could also be accomplished using *lambda expressions*. Those interested can read more about the differences

More expression-bodied members

C# 6 introduced expression-bodied members for member functions, and read-only properties. C# 7 expands the allowed members that can be implemented as expressions. In C# 7, you can implement *constructors*, *finalizers*, and `get` and `set` accessors on *properties* and *indexers*. The following code shows examples of each:

C# Copy

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");
```

```
private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

Note

This example does not need a finalizer, but it is shown to demonstrate the syntax. You should not implement a finalizer in your class unless it is necessary to release unmanaged resources. You should also consider using the SafeHandle class instead of managing unmanaged resources directly.

These new locations for expression-bodied members represent an important milestone for the C# language: These features were implemented by community members working on the open-source Roslyn project.

Throw expressions

In C#, `throw` has always been a statement. Because `throw` is a statement, not an expression, there were C# constructs where you could not use it. These included conditional expressions, null coalescing expressions, and some lambda expressions. The addition of expression-bodied members adds more locations where `throw` expressions would be useful. So that you can write any of these constructs, C# 7 introduces *throw expressions*.

The syntax is the same as you've always used for `throw` statements. The only difference is that now you can place them in new locations, such as in a conditional expression:

C# Copy

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "New name
must not be null");
}
```

This features enables using throw expressions in initialization expressions:

C# Copy

```
private ConfigResource loadedConfig = LoadConfigResourceOrDefault() ??
    throw new InvalidOperationException("Could not load config");
```

Previously, those initializations would need to be in a constructor, with the throw statements in the body of the constructor:

C# Copy

```
public ApplicationOptions()
{
    loadedConfig = LoadConfigResourceOrDefault();
    if (loadedConfig == null)
        throw new InvalidOperationException("Could not load config");

}
```

Note

Both of the preceding constructs will cause exceptions to be thrown during the construction of an object. Those are often difficult to recover from. For that reason, designs that throw exceptions during construction are discouraged.

Generalized async return types

Returning a `Task` object from async methods can introduce performance bottlenecks in certain paths. `Task` is a reference type, so using it means allocating an object. In cases where a method declared with the `async` modifier returns a cached result, or completes synchronously, the extra allocations can become a significant time cost in performance critical sections of code. It can become very costly if those allocations occur in tight loops.

The new language feature means that async methods may return other types in addition to `Task`, `Task<T>` and `void`. The returned type must still satisfy the async pattern, meaning a `GetAwaiter` method must be accessible. As one concrete example, the `ValueTask` type has been added to the .NET framework to make use of this new language feature:

C# Copy

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
```

```
    return 5;
}
```

Note

You need to add the pre-release NuGet package `System.Threading.Tasks.Extensions` in order to use `ValueTask` in Visual Studio 15 Preview 5.

A simple optimization would be to use `ValueTask` in places where `Task` would be used before. However, if you want to perform extra optimizations by hand, you can cache results from async work and reuse the result in subsequent calls. The `ValueTask` struct has a constructor with a `Task` parameter so that you can construct a `ValueTask` from the return value of any existing async method:

C# Copy

```
public ValueTask<int> CachedFunc()
{
    return (cache) ? new ValueTask<int>(cacheResult) : new
ValueTask<int>(LoadCache());
}
private bool cache = false;
private int cacheResult;
private async Task<int> LoadCache()
{
    // simulate async work:
    await Task.Delay(100);
    cacheResult = 100;
    cache = true;
    return cacheResult;
}
```

As with all performance recommendations, you should benchmark both versions before making large scale changes to your code.

Numeric literal syntax improvements

Misreading numeric constants can make it harder to understand code when reading it for the first time. This often occurs when those numbers are used as bit masks or other symbolic rather than numeric values. C# 7 includes two new features to make it easier to write numbers in the most readable fashion for the intended use: *binary literals*, and *digit separators*.

For those times when you are creating bit masks, or whenever a binary representation of a number makes the most readable code, write that number in binary:

C# Copy

```
public const int One =  0b0001;
public const int Two =  0b0010;
public const int Four = 0b0100;
public const int Eight = 0b1000;
```

The `0b` at the beginning of the constant indicates that the number is written as a binary number.

Binary numbers can get very long, so it's often easier to see the bit patterns by introducing the `_` as a digit separator:

C# Copy

```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

The digit separator can appear anywhere in the constant. For base 10 numbers, it would be common to use it as a thousands separator:

C# Copy

```
public const long BillionsAndBillions = 100_000_000_000;
```

The digit separator can be used with `decimal`, `float` and `double` types as well:

C# Copy

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio =
1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

Taken together, you can declare numeric constants with much more readability.