Number one on the list of future C# features is Nullable Reference Types. We first covered this last year, but to briefly recap: all reference variables, parameters, and fields are going to be non-nullable by default. Then, like value types, if you want anything to be nullable you must explicitly indicate that by appending a question mark (?) to the type name.

This will be an optional feature and the current thought is nullable reference types will be turned off for existing projects that are being upgraded to C# 8. For new projects, Microsoft is inclined to turn the feature on by default.

The warnings will be further subdivided into potential errors and merely cosmetic warnings. For example, if p.MiddleName is a string?, then this line would be a cosmetic warning:

```
string middleName = p.MiddleName;
```

Since the danger doesn't occur until the value is dereferenced, this assignment to a local variable isn't really a problem. So, you can disable the warning on legacy code to cut down on the number of false positives.

Likewise, libraries predating this feature won't trigger warnings because the compiler doesn't know whether or not a given parameter should be treated as nullable.

A preview of nullable reference types is available on GitHub.

Switch Expressions

Switch blocks are commonly used to simply return a single value. In this common scenario, the syntax is quite verbose compared to what is being done. Consider this example using pattern matching:

```
static string M(Person person)
{
    switch (person)
    {
        case Professor p:
            return $"Dr. {p.LastName}";
        case Studen s:
            return $"{s.FirstName} {s.LastName} ({s.Level})";
        default:
            return $"{person.FirstName} {person.LastName}";
    }
}
```

Under the new proposal, the repetitive case and return statements can be eliminated. This results in this newer, more compact syntax.

```
static string M(Person person)
{
```

```
    return person switch
    {
        Professor p => $"Dr. {p.LastName}",
        Student s => $"{s.FirstName} {s.LastName} ({s.Level})",
        _ => $"{person.FirstName} {person.LastName}"
    };
}
```

The exact syntax is still under debate. For example, it hasn't been decided if this feature will use => or : to separate the pattern expression from the return value.

Property Pattern Matching

Currently it is possible to perform property level pattern matching via the when clause.

```
case Person p when p.LastName == "Cambell" : return $"{p.FirstName} is not enrc
```

With "property patterns" this is reduced to

```
case Person { LastName: "Cambell"} p : return $"{p.FirstName} is not enrolled";
```

A small gain, but it does make the code clearer and removes a redundant instance of the variable name.

Extracting Properties from Patterns

Taking it a step further, you can also declare variables in the pattern.

```
case Person { LastName: "Cambell", FirstName: var fn} p : return $"{fn} is not
```

Deconstructors in Patterns

A deconstructor is used to break down an object into its constituent parts. It is primarily used for multiple assignment from a tuple. With C# 7.3, you can also use deconstruction with pattern matching.

In this next example, the Person class deconstructs to {FirstName, MiddleName, LastName}. Since we are not using MiddleName, an underscore is used as a placeholder for the skipped property.

```
case Person ( var fn, _, "Cambell" ) p : return $"{fn} is not enrolled";
```

Recursive Patterns

For our next pattern, let us say the class Student deconstructs into {FirstName, MiddleName, LastName, Professor}. We can decompose both the Student object and its child Professor object

at the same time.

```
case Student ( var fn, _, "Cambell", var (_, _, ln) ) p : return $"{fn} is enrc
```

In this line of code, we:

1. Extracted student.FirstName into fn
2. Skipped student.MiddleName
3. Matched student.LastName to “Cambell”
4. Skipped student.Professor.FirstName and student.Professor.MiddleName
5. Extracted student.Professor.LastName into ln

A preview of recursive pattern matching is available on GitHub.

Index Expressions

A range expression eliminates much of the verbose (and error prone) syntax for dealing with array-like data structures.

The hat operator (^) counts from the end of the list. For example, to get the last value in a string you could write,

```
var lastCharacter = myString[myString.Length-1];
```

or simply

```
var lastCharacter = myString[^1];
```

This works with computed values such as

```
Index nextIndex = ^(x + 1);
var nextChar = myString[nextIndex]
```

Range Expressions

Closely related to index expressions are range expressions, denoted with the (..) operator. Here is a simple example that gets the first three characters in a string.

```
var s = myString.Substring[0..2];
```

This can be combined with an index expression. In the next line, we skip the first and last character.

```
var s = myString.Substring(1..^1);
```

This also works with Span<T>.

```
Span<int> slice = myArray[4…8];
```

Note the first number is inclusive and the second number is exclusive. So, the variable slice would have elements 4 thru 7.

To get a slice with all elements starting at an index, there are two options.

```
Span<int> rest = myArray[8..];
Span<int> rest = myArray[8..^0];
```

Likewise, you can omit the first index.

```
Span<int> firstFour = myArray[..4];
```

In case you are wondering, this syntax was heavily inspired by Python. The major difference is C# can't use -1 for indexing from the end of an array because that already has a meaning in .NET arrays. So instead we get the ^1 syntax.

A preview of indexes and ranges is available on GitHub.

IAsyncDisposable

As the name implies, this interface allows objects to expose a DisposeAsync method. In conjunction with this is the new syntax "using async" which works just like a normal using block with the added ability to await on a dispose call.

Asynchronous Enumerators

Like IEnumerable<T>, IAsyncEnumerable<T> will allow you to enumerate a finite list of unknown length. The matching enumerator looks slightly different though. It exposes two methods:

```
Task<bool> WaitForNextAsync();
T TryGetNext(out bool success);
```

An interesting feature of this interface is it allows you to read data in batches. You call TryGetNext for each item in the batch. When that returns success=false, you then call WaitForNextAsync to fetch a new batch.

The reason this is important is most data comes to the application in either batches or streamed over the network. When you call TryGetNext, the data will already be available most of the time. Allocating a Task object would be wasteful, but if you do run out of data in the input buffer, you still want to be able to asynchronously wait for more.

Ideally you wouldn't use these interfaces directly very often. Instead Microsoft would like you to use the "foreach await" syntax known as an asynchronous iterator, which we previewed last year. This will handle calling the synchronous or asynchronous method as necessary.

Default Interface Methods

This controversial feature inspired by Java is still being considered for C# 8. Briefly, it allows you to evolve an interface by adding new methods with matching implementations. This way the new

method won’t break backwards compatibility.

Records

Records are a syntax for quickly creating immutable classes. We first saw this proposal in 2014. The current example looks like this:

```
class Person(string FirstName, string LastName);
```

Basically, it’s a class defined entirely by a constructor’s signature. All of the properties and methods you would expect from an immutable class are automatically generated.