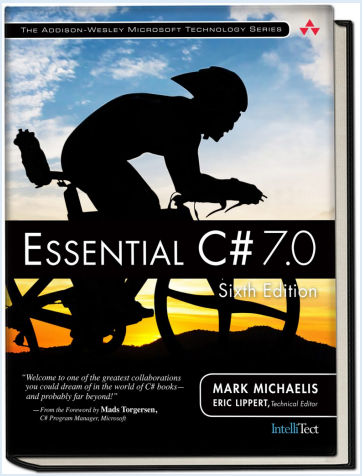
Essential C# 8.0

Mark Michaelis

About Us

Mark Michaelis

Chief Technical Architect,
Author, Trainer



Microsoft® Most Valuable Professional

Microsoft® Regional Director

Professional Scrum Master II

If you were king, what would you do with C#?

Nullable Reference Types

- Invoking a member on a null value will issue a `System.NullReferenceException` exception, and every invocation that results in a `System.NullReferenceException` in production code is a bug.
- With nullable reference types we “fall in” to doing the wrong thing rather than the right thing. (The “fall in” action is to invoke a reference type without checking for null.)
- There’s an inconsistency between reference types and value types (following the introduction of Nullable<T>) in that value types are nullable when decorated with “?” (for example, int? number); otherwise, they default to non-nullable.
- It’s not possible to run static flow analysis to check all paths regarding whether a value will be null before dereferencing it, or not.
- There’s no reasonable syntax to indicate that a reference type value of null is invalid for a particular declaration.
- There’s no way to decorate parameters to not allow null

What to do about it?

Provide syntax to expect null:

- Enable the developer to explicitly identify when a reference type is expected to contain nulls—and, therefore, not flag occasions when it's explicitly assigned null.

Make default reference types expect non-nullable:

- Change the default expectation of all reference types to be non-nullable, but do so with an opt-in compiler switch rather than suddenly overwhelm the developer with warnings for existing code.

Decrease the occurrence of NullReferenceExceptions:

- Reduce the likelihood of NullReferenceException exceptions by improving the static flow analysis that flags potential occasions where a value hasn't been explicitly checked for null before invoking one of the value's members.

Enable suppression of static flow analysis warning:

- Support some form of "trust me, I'm a programmer" declaration that allows the developer to override the static flow analysis of the complier and, therefore, suppress any warnings of a possible NullReferenceException.

Demo

Conclusion: Nullable Reference Type

- Warning you to remove a null assignment to a non-nullable type potentially eliminates a bug because a value is no longer null when it shouldn't be.
- Alternatively, adding a nullable modifier improves your code by being more explicit about your intent.
- Over time the impedance mismatch between nullable updated code and older code will dissolve, decreasing the `NullReferenceException` bugs that used to occur.
- The nullability feature is off by default on existing projects so you can delay dealing with it until a time of your choosing. In the end you have more robust code. For cases where you know better than the compiler, you can use the ! operator (declaring, "Trust me, I'm a programmer.") like a cast.
- Nullable types don't have any semantic impact, they only issue warnings.

Syntax to Expect Null

```
class Address
{
    public string Street1 { get; }
    public string? Street2 { get; }
    public string City { get; }
    public string Zip { get; }
    public string Country { get; }

}
```

Show potential fixes (Alt+Enter or Ctrl+.)

Index

```
// Initialize new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8}
int[] array = Enumerable.Range(0, 9).ToArray();

lastItem = array[(array.Length - 1)];
Assert.AreEqual(8, lastItem);

lastItem = array[new Index(1, true)];
Assert.AreEqual(8, lastItem);

lastItem = array[^1];
Assert.AreEqual(8, lastItem);
```

Index

```
Span<int> span;

span = array[Range.Create(4, new Index(2, true))];
Assert.AreEquivalent(new int[]{4, 5, 6}, span);

span = array[4..^2];    // array[Range.Create(4, new Index(2, true))]
Assert.AreEquivalent(new int[]{4, 5, 6}, span);

span = array[..^3];     // array[Range.ToEnd(new Index(3, true))]
Assert.AreEquivalent(new int[]{0, 1, 2, 3, 4, 5}, span);

span = array[2..];      // array[Range.FromStart(2)]
Assert.AreEquivalent(new int[]{2, 3, 4, 5, 6, 7, 8}, span);

span = array[Range.All()]; // array[Range.All()]
Assert.AreEquivalent(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8}, span);
```

System.Span<T>, System.Index, System.Range

Span<T>
Generic Structure

- Fields
  - _length As int
- Properties
  - Empty As Span<T>
  - IsEmpty As bool
  - Length As int
- Methods
  - Equals() As bool
  - GetEnumerator() As Enumerator
  - GetHashCode() As int
  - implicit operator Span<T>() As Span<T> (+ 1 overload)
- Nested Types

Range
Structure

- Properties
  - End As Index
  - Start As Index
- Methods
  - All() As Range
  - Create() As Range
  - FromStart() As Range
  - Range()
  - ToEnd() As Range

Index
Structure

- Fields
  - _value As int
- Properties
  - FromEnd As bool
  - Value As int
- Methods
  - implicit operator Index() As Index
  - Index()

Switch Expressions

Switch expressions are enabled:

```
return switch
{
    Professor p => $"{p.LastName}"
}
```

Property Patterns

```
person switch
{
        Person { LastName: "Montoya“, FirstName: var firstName }
        null => "the value is null"
}
```

“string” wouldn’t match nullable values. (string? not allowed). var is the nullable version

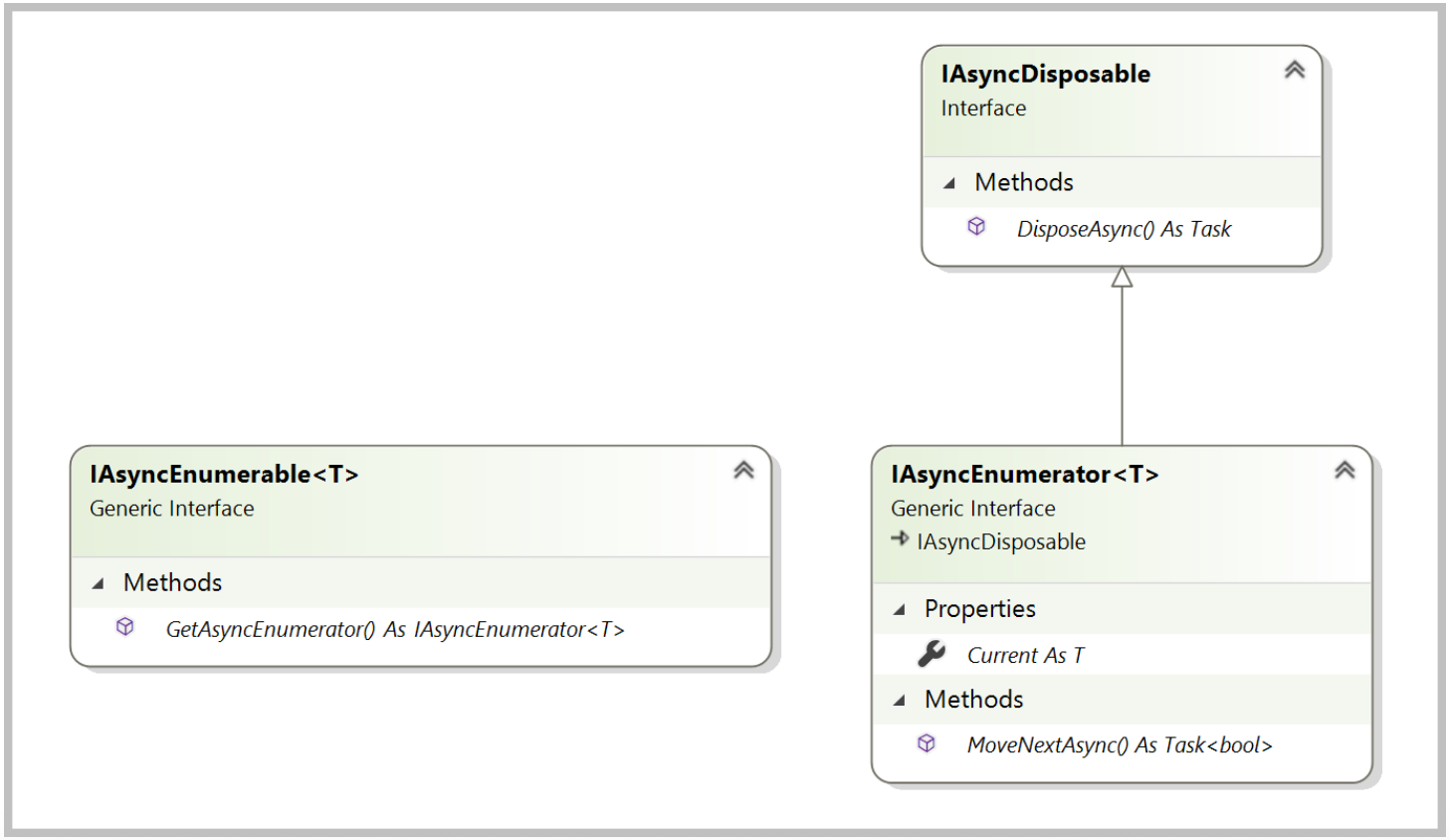
Type pattern never matches a null.

Exception is thown if there is fall through

Be sure that if you use a deconstructor, the order needs to be well known or obvious.

Async Streams

Enabling LINQ Over Events



IntelliTect

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(
                out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

IntelliTect

With Syntax

```
foreach await (T item in enumerable)
{
    Use(item);
}
```

Default Interfaces

```
class ITraceable
{
    static public int IndentationCount
        { get; set; }

    public string GetMessage() =>
        this.ToString();
}
```

- Explicit access modifiers would be permissible: `private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `override`, `sealed`, `static`, `extern` (the default is `public`).
- You could not have fields.
- Static methods, properties, indexers, and events would also be allowable.

Enhanced Pattern Matching

```
return switch (person)
 {
    Professor(_, var lastName, var subject) item =>
                $"Dr. { lastName} teaching {subject}",
    Student { FirstName: var firstName,
        Advisor { LastName: var advisorLastName } } =>
                $"",
    (string fistName, _)
        { EnrollmentStatus: EnrollmentStatus.Enrolled } =>
                $"",
    { } => "Enrollment has passed.  See you next year.",
    null => "Oh No!!"
}
```

IntelliTect

What else…?

Async Streams:

- Enables LINQ over events

Make default reference types expect non-nullable:

- Change the default expectation of all reference types to be non-nullable, but do so with an opt-in compiler switch rather than suddenly overwhelm the developer with warnings for existing code.

Decrease the occurrence of NullReferenceExceptions:

- Reduce the likelihood of NullReferenceException exceptions by improving the static flow analysis that flags potential occasions where a value hasn't been explicitly checked for null before invoking one of the value's members.

Enable suppression of static flow analysis warning:

- Support some form of "trust me, I'm a programmer" declaration that allows the developer to override the static flow analysis of the complier and, therefore, suppress any warnings of a possible NullReferenceException.

Mark Michaelis
mark@IntelliTect.com
@IntelliTect, fb.com/IntelliTect

IntelliTect
INTELLIGENT.TECHNICAL.ARCHITECTURE.TRAINING

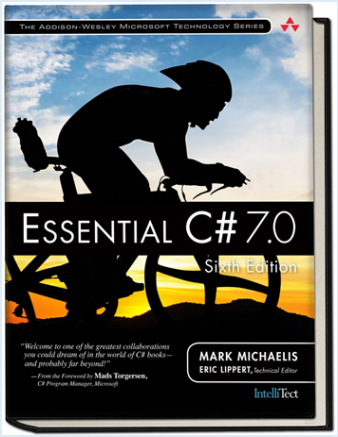
About Us

Mark Michaelis

Chief Technical Architect,
Author, Trainer

mark@IntelliTect.com

Twitter: @MarkMichaelis,
fb.com/MarkMichaelis



Microsoft® Most Valuable Professional

Microsoft Regional Director

Professional Scrum Master II

IntelliTect @IntelliTect, fb.com/IntelliTect