WHAT'S NEW IN C# 7.X AND C# 8

Philip Japikse (@skimedic)
skimedic@outlook.com
www.skimedic.com/blog
Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, PSM II, PSD
Consultant, Teacher, Writer

Phil.About()

- Director of Consulting/Chief Architect
- Author: Apress.com (http://bit.ly/apressbooks)
- Speaker: http://www.skimedic.com/blog/page/Abstracts.aspx
- Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, PSM II, PSD
- Founder, Agile Conferences, Inc.
  - http://www.cincydeliver.org
- President, Cincinnati .NET User's Group

Building Web Applications with Visual Studio 2017

Using .NET Core and Modern JavaScript Frameworks

Philip Japikse
Kevin Grossnicklaus
Ben Dewey

Apress

Pro C# 7.0

With .NET and .NET Core

Andrew Troelsen
Philip Japikse

Apress

C# 7.0

LOCAL FUNCTIONS

- Allow for declaring methods inside the context of another method
  - Designed for methods only called from one place

- Use cases:
  - Iterator and/or Async operations
    - Exceptions only occur when iterated or awaited
  - Local encapsulation

TUPLES

- Lightweight data structure containing more than one field/member
- Can be better than creating a class or a struct for single use
- Fields are not validated
- Cannot define your own methods

- NOTE: Improved in 7.1 and 7.3

TUPLES NOTES

- Available before C# 7, but were inefficient and had no language support
- In C# 7, might need the System.ValueTuple depending on .NET F/X Version
- Tuples are updated again with C# 7.1 and 7.3

DECONSTRUCTING TUPLES

- Unpackages Tuples into separate variables
- Any .NET type can be deconstructed be creating a Deconstruct method.
  - Must provide out parameters for each property to deconstruct
  - Assign a tuple to the deconstruct method
    - Names on the left override names on the right

OUT VARIABLES

- No need to declare a variable before using it as an out parameter
- Support explicit and implicit declarations
- Declared variable “leaks” out of if statements in the Try pattern

```
bool canParse = int.TryParse("123", out int result);
bool canParse2 = int.TryParse("123", out var result2);

if (bool.TryParse("true", out var boolResult))
{
    //Do something
}

return boolResult;
```

PATTERN MATCHING

- Allows implementing method dispatch on properties other than the type of the object
- Pattern matching supports the is and switch statements

PATTERN MATCHING WITH IS

- Allows assignment of variable while type checking

```
if (item is int val)
if (item is IEnumerable<object> subList)
If (item is MyClass m)
```

PATTERN MATCHING IN SWITCH STATEMENTS

- Allows type checking and assignment of variable in addition to value checking
  - “When” provides additional filtering
  - Order of the statements matters when using pattern matching

```
case 0:
case int val:
case IEnumerable<object> subList when subList.Any():
case IEnumerable<object> subList:
case var i when i is string:
case null:
default:
```

DISCARDS

- Allows ignoring return (or out) parameter
- Discard variable is write-only and named "_"
- Useful for Deconstructing tuples
  - Calling methods with out parameters
  - Pattern matching with is and switch operators
  - Standalone when you just want to discard the assignment

DISCARD WATCH OUTS

- "_" was valid before it became a discard symbol
- If declared before using discard:
  - Can update the original variable value or throw due to type clash

MORE EXPRESSION BODIES MEMBERS

- Expression bodied members were introduced in C# 6
- C# 7 expanded the available list to include:
  - Constructors
  - Finalizers
  - get/set accessors on properties and indexers

THROW EXPRESSIONS

- Throw has always been a statement, preventing use in:
  - Conditional expressions, null coalescing expressions, some lambdas
- The expansion of Expression-Bodied Members adds more locations through the use of throw expressions

GENERALIZED ASYNC RETURN TYPES

- Task is a reference type. Returning Task<int> *can* introduce performance issues
- Async may return other types as long as type satisfies the async pattern (GetAwaiter is accessible)
- One concrete type (ValueTask) was added to the .NET f/x to use this feature

```
public async ValueTask<int> GeneralAsyncReturnTypes()
{
    await Task.Delay(100);
    return 5;
}
```

NUMERIC LITERAL SYNTAX IMPROVEMENTS

- Binary Literals begin with 0b
- Digit separators help readability

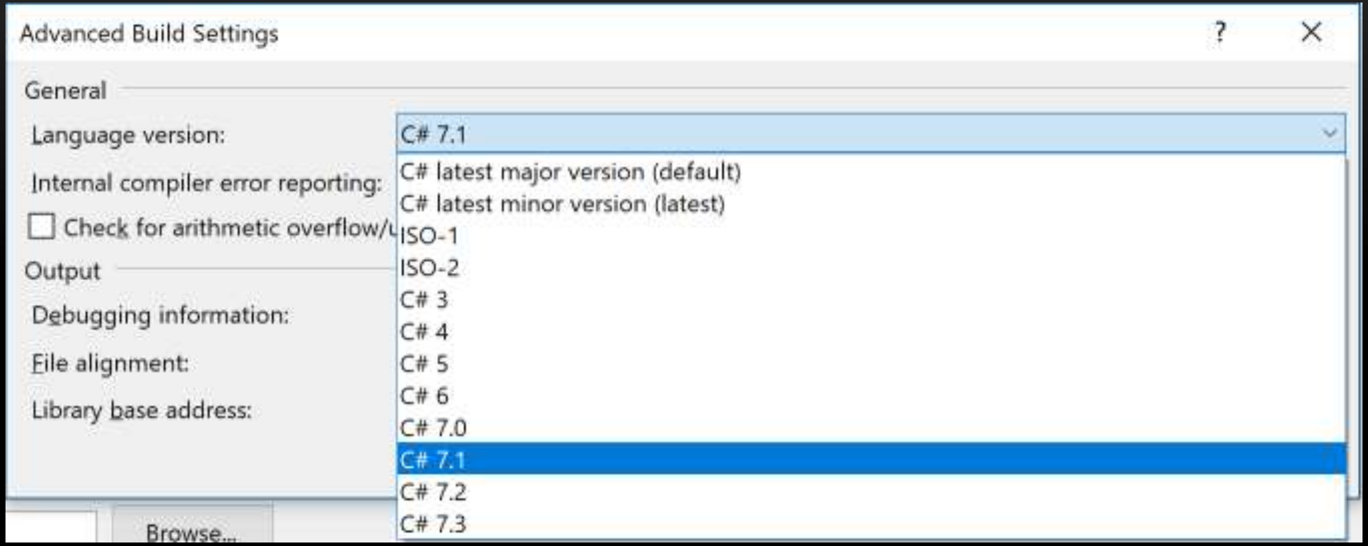
```
public const int Sixteen = 0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;

public const long BillionsAndBillions = 100_000_000_000;
public const double AvogadroConstant = 6.022_140_857_747_474e23;
```

- Note: Improved in 7.2

C# 7.1

ENABLING MINOR VERSIONS

- Project Properties:
  - Build -> Advanced



Advanced Build Settings

General

Language version: C# 7.1

Internal compiler error reporting:

Check for arithmetic overflow/u

Output

Debugging information:

File alignment:

Library base address:

Browse...

C# latest major version (default)
C# latest minor version (latest)
ISO-1
ISO-2
C# 3
C# 4
C# 5
C# 6
C# 7.0
C# 7.1
C# 7.2
C# 7.3

- Edit the project file:

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
  <LangVersion>7.1</LangVersion>
</PropertyGroup>

<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
  <LangVersion>7.1</LangVersion>
</PropertyGroup>
```

ASYNC MAIN METHOD

- Allows for using async methods from apps Main method

```
static async Task<int> Main(string[] args)
{
}
//or
static async Task Main(string[] args)
{
}
```

INFERRED TUPLE NAMES

- Variable names on the left are inferred from the variable names used to build the tuple
  - Reserved names (e.g. ItemX) are not used
  - Duplicates are not allowed
- Explicit names override inferred names
- Left side still wins over the right

```
var alpha = "a";
var beta = "b";
var letters = (alpha, beta);
var firstLetter2 = letters.alpha;
var secondLetter2 = letters.beta;
```

- NOTE: Improved in 7.3

DEFAULT LITERAL EXPRESSIONS

- Initialize a variable to its default value
  - Reference types => null
  - Numeric => 0
  - Enum => 0
  - Bool => false
  - Struct => all value types to default and all ref types to null
  - Nullable => HasValue = false, Value = undefined
  - Generics => T t = default(T)

C# 7.2

LEADING UNDERSCORES IN NUMERIC LITERALS

- This is a small tweak to the Literal improvements in 7.0
- Hex and Binary can start with "_"

```
public const int Sixteen = 0b_0001_0000;
public const int ThirtyTwo = 0b_0010_0000;
public const int SixtyFour = 0b_0100_0000;
public const int OneHundredTwentyEight = 0b_1000_0000;
```

- Note: R# must be enabled for 7.2 improvements.
  - Right Click, Edit Project Item Properties, C# Language Level

THE IN KEYWORD

- Specifies an arg is passed byref but not modified by the called method
- Stack variables are copied unless modified with:
  - out: Called method sets the value
  - ref: Called method *may* set the value of the argument
  - in: Does not modify the value of the argument

- Calling method without in can cause defensive copying
  - But allows for gradual modification of your code base

CONDITIONAL REF EXPRESSIONS

- Conditional Expression can return a reference instead of a value

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

PRIVATE PROTECTED ACCESS MODIFIER

- Can now declare a member private AND protected.
  - Can only be accessed by derived classes *declared* in the same assembly

- Protected internal allows access to derived classes or classes in the same assembly
  - InternalsVisibleTo can expand the reach for protected internals

NON-TRAILING NAMED ARGUMENTS

- Can now mix named and positional arguments as long as they are in order
- I’m not going to show an example…I think it’s a code smell

C# 7.3

ADDITIONAL GENERIC CONSTRAINTS

- Can now use Enum or Delegate

```
public class UsingEnum<T> where T : System.Enum { }
public class UsingDelegate<T> where T : System.Delegate { }
public class Multicaster<T> where T : System.MulticastDelegate { }
```

- Can also use unmanaged (enforces struct)

```
class UnManagedWrapper<T> where T : unmanaged
{ }
```

ENHANCEMENTS TO EXISTING FEATURES

- Use == and != with tuples
- Attach attributes to backing fields of auto-properties
- Additional locations for expression variables

TUPLE EQUALITY

- Tuples support the == and != operators
- Comparisons perform lifting and widening conversions where needed
- Names aren’t used for comparison
  - If names don’t match, compiler warning CS8383 is raised

ATTACH ATTRIBUTES TO BACK FIELDS FOR AUTO PROPS

- Allows mixing of Auto Properties and custom Attributes for get/set
  - E.g. WPF INotifyPropertyChanged

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

EXPRESSION VARIABLES IN INITIALIZERS AND LINQ QUERIES

- You can use out variables in field initializers, property initializers, constructor initializers, and query clauses.

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

C# 8

C# 8 IS TIED TO .NET STANDARD 2.1

- Many of the C# 8.0 language features have platform dependencies. Async streams, indexers and ranges all rely on new framework types that will be part of .NET Standard 2.1.
- For this reason, using C# 8.0 is only supported on platforms that implement .NET Standard 2.1. The need to keep the runtime stable has prevented us from implementing new language features in it for more than a decade.

https://devblogs.microsoft.com/dotnet/building-c-8-0/

A PREVIEW OF C# 8 (FINALIZED 23-SEPT)

- Nullable Reference Types
- Ranges and Indices
- Static local functions
- Using declarations
- Additional Pattern Matching
  - Switch expressions
  - Property Patterns
  - Tuple patterns
  - Positional
- Interface updates
  - Default methods/implementations
- Read-only members (structs)
- Null-coalescing Assignment
- Interpolated Verbatim Strings
- Async Streams

NULLABLE TYPES

NULLABILITY OF TYPES

- Non-nullable – null can’t be assigned to the variable
- Nullable – null can be assigned to the variable
- Oblivious – Pre C# 8 state
- Unknown – Type parameters where constraints don’t tell the compiler the nullability

NULLABLE PROJECT SETTINGS

```
<Nullable>enable</Nullable>
```

Project Setting	Nullable Annotation Context (?)	Nullable Warning Annotation Context	Reference Types	Nullable Warnings
enable	Enabled	Enabled	Specifically defined nullable or not nullable	Enabled
warnings	Disabled	Enabled	Oblivious	Enabled
annotations	Enabled	Disabled	Oblivious	Enabled
disable (default)	Disabled	Disabled	Oblivious	Disabled

NULLABLE COMPILER DIRECTIVES

```
#nullable enable
```

Directive	Nullable Annotation Context (?)	Reference Types	Nullable Warnings
enable	Enabled	Specifically defined nullable or not nullable	Enabled
disable	Disabled	Oblivious	Enabled
restore	Project Setting	Project setting	Enabled

```
#pragma warning <enable || disable || restore> nullable
```

NULLABLE ATTRIBUTES AND CONSTRAINTS

- Preconditions (input conditions):
  - AllowNull: A non-nullable input argument may be null.
  - DisallowNull: A nullable input argument should never be null.
- Postconditions (output conditions):
  - MaybeNull: A non-nullable return value may be null.
  - NotNull: A nullable return value will never be null.

NULLABLE ATTRIBUTES AND CONSTRAINTS

- Conditional Post-conditions:
  - MaybeNullWhen: A non-nullable out or ref argument may be null when the return value satisfies a condition.
  - NotNullWhen: A nullable out or ref argument may not be null when the return value satisfies a condition.
  - NotNullIfNotNull: A return value isn't null if the input argument for the specified parameter isn't null.

NULLABILITY AND GENERICS

- Must specify struct or class constraint to the generic

DEMO

DEMO

Nullable Types in C# 8

INDICES AND RANGES

INDICES AND RANGES

- Provide a succinct syntax for specifying subranges in an array, string, Span<T>, or ReadOnlySpan<T>.
- Two new types:
  - System.Index => an index into a sequence.
  - System.Range => a sub range of a sequence.
- Two new operators:
  - The ^ operator, which specifies that an index is relative to the end of the sequence.
  - The Range operator (..), which specifies the start and end of a range as its operands.

INDICES AND RANGES IN PRACTICE

- Index counts back from end of sequence
  - sequence[^0] == sequence[sequence.Length]
  - sequence[^n] == sequence[sequence.Length-n]
- A range specifies the start and end of a range.
  - Start is inclusive, end is not
  - sequence[0..^0] == sequence[0..sequence.Length]
  - sequence[..] == sequence[0..sequence.Length]
  - Either end can be open

DEMO

DEMO

Indices and Ranges

STATIC LOCAL FUNCTIONS

STATIC LOCAL FUNCTIONS

- Static Local Functions don’t reference any variables from the enclosing scope

DEMO

Static Local Functions

USING DECLARATIONS

USING DECLARATIONS

- Variable declaration preceded by the using keyword.
- Compiler will dispose of the variable at the end of the enclosing scope.

USING DECLARATIONS

```
static void Foo(IEnumerable<string> lines)
{
    using var foobar = // new something
    //more code
    // file variable is disposed here
}
```

```
static void Bar (IEnumerable<string> lines)
{
    using (var foobar = //new something)
    {
        //more code
    } // file variable is disposed here
}
```

DEMO

DEMO

Static Local Functions

MORE PATTERNS, MORE PLACES

- Pattern matching provides functionality across related but different kinds of data.
- C# 7.0 introduced syntax for type patterns and constant patterns with the is expression and the switch statement.
- C# 8.0 allows you to use more pattern expressions in more places in your code as well as recursive patterns.
  - A recursive pattern is simply a pattern expression applied to the output of another pattern expression.

SWITCH EXPRESSIONS

- The variable comes before the switch keyword.
- The case and : elements are replaced with =>.
- The default case is replaced with a _ discard.
- The bodies are expressions, not statements.
- Must either produce a value or throw an exception.
  - Discards typically are used to throw an exception.

PROPERTY PATTERNS

- Property Patterns
  - Enable matching on properties of the object examined.
- Tuple Patterns
  - Switch based on multiple values expressed as a tuple.
- Positional Patterns
  - Types with Deconstruct methods can use positional patterns to inspect properties of the object and use those properties for a pattern.

DEMO

DEMO

Patterns

INTERFACES

DEFAULT IMPLEMENTATIONS

- Methods defined in an interface
  - Can be overridden by implementors
  - Allow for expanding interfaces without breaking existing code

DEMO

DEMO

Default Implementations

STRUCTS

READ ONLY MODIFIER

- Read Only Modifier
  - Indicates that the member does not modify state.
  - More granular than the read only modifier to a struct declaration
- Disposable [Readonly] Ref Structs
  - Must add a Dispose method

DEMO

Read Only Modifier

NULL COALESCING ASSIGNMENT
ENHANCEMENT OF INTERPOLATED VERBATIM STRINGS

MISCELANEOUS ENHANCEMENTS

- Null Coalescing Assignment Operator (??=)
  - Assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null.
  - i ??= 17
- Interpolated Verbatim String
  - Order of $ and @ no longer matters

ASYNC STREAMS

ASYNC STREAMS

- A method that returns an asynchronous stream has three properties:
  - It's declared with the async modifier.
  - It returns an IAsyncEnumerable<T>.
  - The method contains yield return statements to return successive elements in the asynchronous stream.

DEMO

DEMO

Async Streams

Contact Me

skimedic@outlook.com
www.skimedic.com/blog
www.twitter.com/skimedic

http://bit.ly/skimediclyndacourses
http://bit.ly/apressbooks

www.hallwayconversations.com

Questions?

Thank You!

Get the code: https://github.com/skimedic/presentations