What's New in C# 13

Dmitri Nesteruk @dnesteruk

Overview

- Params collections
- New Lock type
- Ref struct improvements
- Ref/unsafe in iterators and async method
- Assorted improvements

Params

- `params` allows variable number of arguments
- Uses: instead of

```
op == "AND" || op == "OR" || op == "XOR"
```

  … *or* …

```
new[]{"AND", "OR", "XOR"}.Contains(op);
```

- We can write

```
op.IsOneOf("AND", "OR", "XOR");
```

-
  ```
  public static bool IsOneOf<T>(this T self,
    params T[] values) => values.Contains(self);
  ```

C#13 Params

- In C#13, params can be used with
  - 1-dimensional array type `T[]`
  - Span types (`Span<T>`, `ReadOnlySpan<T>`)
  - Types with a create method (see CollectionBuilderAttribute)
- Also supports some interfaces:
  - `IEnumerable<T>`
  - `ICollection<T>/IReadOnlyCollection<T>`
  - `IList<T>/IReadOnlyList<T>`
- Struct or class that implements IEnumerable
  - Type has a constructor and an `Add()` method

Example

```
public static int Test(params IList<int> values)
{
  return values.Sum();
}

Test(1,2,3);
```

Compiler-Generated Code

```
int num = 3;
List<int> list = new List<int>(num);
CollectionsMarshal.SetCount(list, num);
Span<int> span = CollectionsMarshal.AsSpan(list);
int num2 = 0;
span[num2] = 1;
num2++;
span[num2] = 2;
num2++;
span[num2] = 3;
num2++;
Test(list);
```

It gets worse!

- Concise code generation is only done for IList/List
- ICollection<T> — uses List<T>
- Collection<T> — very concise, just some Add() calls
- IReadOnlyList<T>, IReadOnlyCollection<T>, IEnumerable<T>
  - Uses compiler-generated ReadOnlyArray<T>
  - This type implements all params-interfaces at the same time
  - Synthetically initialized from ordinary array
- Use sharplab.io for more insights ☺

Synchronization in .NET

- Traditional uses of locks:

```
private readonly object padlock = new object();
lock(padlock) { … }
```

- Uses System.Threading.Monitor

-
  ```
  try { Monitor.Enter(tempLock, ref lockTaken);
           /* Your code here */
  } finally {
    if (lockTaken) Monitor.Exit(tempLock);
  }
  ```

New Lock Type

- Why?
  - Monitor has performance overhead and lack of flexibility (no timeouts, cannot use `using`)
  - New lock is has improved performance (up to +25%), more readable code, backwards compatible
- New type: System.Threading.Lock
- `x.EnterScope()` returns a disposable ref struct
- `x.TryEnter(timeout)` waits to enter if possible
- Special support when used inside `lock(x)` statement
  - Equivalent to `using(x.EnterScope())`

ref and unsafe in iterators and async methods

- Before C# 13, iterator methods (yield return) and async methods
  - Could not declare local `ref` variables
  - Could not have an unsafe context
- Now, `async` methods can declare `ref` local variables or variables of `ref` struct types
- These cannot be accessed across
  - An `await` boundary
  - A `yield return` boundary
- Unsafe context is allowed in iterator methods
  - `yield return` and `yield break` must be in safe context

ref struct improvements

- ref structs can now be used in generic arguments
- ref structs can implement interfaces (with some caveats)
- ref structs and ref locals can now exist in async methods

More Partial Members

- Partial is now allowed on properties

-
  ```
  partial class Foo {
    public partial int Capacity { get; }
  }
  ```

-
  ```
  partial class Foo {
    public partial int Capacity { get { … } }
  }
  ```
- Same for indexer

  ```
  public partial string this[int index] { … }
  ```

Field keyword

- Preview feature (need to enable)
- Refers to property's backing field

```
public string Name
{
  get;
  set => SetAndRaiseIfChanged(ref field, value);
}
```

- Potentially breaking change

Minor Features

- OverloadResolutionPriorityAttribute
- \e = literal for ESCAPE character
  - Used in e.g. ANSI codes for formatting
  - `Console.WriteLine("\e[1mThis is bold text\e[0m");`
- Method group natural type
- Implicit index access ^ in initializers

```
new Foo {
  bar = { [^1] = 0 }
};
```

Thank You!

End of Video

- Enjoy C#!
- 𝕏 @dnesteruk