Extension Members

Methods, Properties, Operators, …

Behavioral Extensions

- You need to extend a type's behavior
  - Modify the type (requires source code, intrusive, breaks OCP)
  - Extend the type (must be non-`sealed`, support LSP)
  - Default interface implementations (slightly intrusive)
  - Use extension methods
- External augmentation of type is normal
  - Factory/Builder pattern — associated construction methods
  - Decorator pattern — additional behavior (but also storage etc.)
- Big demand for modifying *more* of both library and own data types

Extension Method Applicability

- Extension method works on specific types
  - Library and user-defined
- On a generic type `<T>`
  - Possibly constrained
- C# language-specific constructs (e.g., `ValueTuple`)

Extension Methods on Generic Types

```
public static T DeepClone<T>(this T obj)
{
  using (var ms = new MemoryStream())
  {
    var formatter = new BinaryFormatter();
    formatter.Serialize(ms, obj);
    ms.Position = 0;

    return (T) formatter.Deserialize(ms);
  }
}
```

Extension Methods on ValueTuple

```
public static DateTime dmy
   (this (int d, int m, int y) dmy)
   => new(dmy.y, dmy.m, dmy.d);

// usage
var when = (11, 12, 2025).dmy();
```

Extension Blocks

- C#14 introduces `extension` blocks
- Live in top-level, non-generic, static classes
- Allowed extensions
  - Methods (similar to extension methods)
  - Properties (interface, *not* backing fields)
  - Operators (not on `static` classes!, cannot choose visibility, cannot choose static/non-static)
- Define a set of extensions on
  - An instance of some type
  - An entire type (static)

Extension Block Example

```
public static class Extensions
{
  extension(DateTime)
  {
    public static DateTime Tomorrow =>
      DateTime.Now.AddDays(1);
  }

  extension(DateTime dt)
  {
    public bool IsWeekend =>
      dt.DayOfWeek is DayOfWeek.Saturday
                   or DayOfWeek.Sunday;

  }
}
```

```
var tomorrow = DateTime.Tomorrow;
```

```
var party = tomorrow.IsWeekend;
```

Type Division Unnecessary

```
public static class Extensions
{
  extension(DateTime dt)
  {
    public bool IsWeekend =>
      dt.DayOfWeek is DayOfWeek.Saturday or DayOfWeek.Sunday;
  }
  extension(DateTime)
  {
    public static DateTime Tomorrow => DateTime.Now.AddDays(1);
  }
}
```

Type Division Unnecessary

```
public static class Extensions
{
  extension(DateTime dt)
  {
    public bool IsWeekend =>
      dt.DayOfWeek is DayOfWeek.Saturday or DayOfWeek.Sunday;
    public static DateTime Tomorrow => DateTime.Now.AddDays(1);
  }
}
```

Some Notes

- Extension blocks do not turn into classes
  - There is no extension-block-type in IL
- Extension members emitted in the containing type
- Attribute-based metadata
  - [Extension], [SpecialName]
- XML comments on extension blocks are *wasted*

Generics in Extension Blocks

```
public static class EnumerableExtensions
{
  extension<T>(IEnumerable<T> enumerable)
  {
    public bool IsEmpty => !enumerable.Any();
    public void Dump()
    {
      foreach (var item in enumerable)
       Console.WriteLine(item);
    }
  }
}
```

Operators in Extension Blocks

```
extension(int[])
{
  public static int[] operator*(int[] a, int n)
  {
    var result = new int[a.Length];
    for (int i = 0; i < a.Length; i++)
      result[i] = a[i] * n;
    return result;
  }
}

int[] data = [1, 2, 3];
Console.WriteLine(
  string.Join(",", data*3)); // 3, 6, 9
```

Operator Block Constraints

- Cannot choose static/non-static context
  - Ops are static, compound = non-static
- Cannot choose visibility (everything is public)
- Cannot define implicit conversions
  - Except bool via true/false
- Can only define on non-static types
- Compound operators on value types must be `ref`

Compound Assignment Overloading

- Can now overload compound assignment
  - `+=, -=, *=, /=`
  - `%=, &=, |=, ^=, <<=, >>=`
- Void, instance method
- Value types must use ref in extension blocks

Compound Assignment Operator

```
class Holder
{
  public int N;

  public void operator +=(int x)
  {
    N += x;
  }
}
```

Compound Assignment in Extension Blocks

```
extension(int[] arr)
{
  public void operator *= (int value)
  {
    for (int i = 0; i < arr.Length; i++)
      arr[i] *= value;
  }
}
```

Comprehensive Example

- Extend `List<T>` so that += means append
- Get it to work on…
- Single item of type T

```
public void operator += (T item)
```

- Any list (or enumerable, etc.)

```
public void operator += (IList<T> items)
```

- Any int from a ValueTuple (weak typing!)

```
public void operator += (ITuple tuple)
```

```
List<int> stuff = [];
stuff += 1;
stuff += [2, 3];
stuff += (4, 5);
Console.WriteLine(string.Join(",", stuff));
// 1,2,3,4,5
```

Practical Tiny Example

```
[McpServerToolType]
public static class WeatherTool
{
  extension(object)
  {
    public static string operator ~ (object obj)
      => JsonSerializer.Serialize(obj);
  }

  [McpServerTool, UsedImplicitly]
  [Description("Gets the current weather for a given city.")]
  public static string GetWeather(string city)
  {
    var info = new WeatherInfo(city, "Average", 90m);
    return ~info;
  }
}
```

MCP Return Pattern

```
return ~ new { . . . }
```

MCP Return Pattern

```
return~new { . . . }
```

Typical MCP Use

```
[McpServerTool,UsedImplicitly]
[Description("…")]
public static string GetSuggestedBreakerSize
  (decimal wireSizeInSquareMillimeters,
   string circuitType = "radial")
{
  if (!circuitType.In("radial", "ring"))
    return ~ new {
      errorMessage = $"{circuitType} is not supported." };
  …
  return ~ new
  {
    suggestedBreakerSize = result,
    circuitType
  };
}
```

Cpp Style

```
extension(TextWriter)
{
  public static TextWriter operator <<
    (TextWriter tw, object any)
  {
    tw.Write(any);
    return tw;
  } // note: cannot define << for Console
}
var name = "Dima";
string endl = Environment.NewLine;
_ = Console.Out << "Hello, " << name
                << '!' << endl;
```

Cpp Style

```
extension<TKey, TValue>(Dictionary<TKey, TValue> dict)
  where TKey : notnull
{
  public ref TValue? at(TKey key)
  {
    return ref CollectionsMarshal.GetValueRefOrAddDefault
      (dict, key, out _);
  }
}
Dictionary<string, int> d = new();
Console.WriteLine($"x = {d.at("x")}"); // x = 0
d.at("x") = 1;
d.at("x") += 2;
Console.WriteLine(string.Join(",",
  d.Select(kv => $"{kv.Key} = {kv.Value}")));
```

Python Style

```
extension(string s)
{
  public static string operator *
    (string s, int n)
    => string.Concat(Enumerable.Repeat(s, n));
}
Console.WriteLine("a" * 3); // aaa
```

Python Style

```
extension<K, V>(IDictionary<K, V>)
{
  public static bool operator true(IDictionary<K, V> dict)
    => return dict.Any();
  public static bool operator false(IDictionary<K, V> dict)
    return !dict.Any(); // symmetry
}

if (dict)
  Console.WriteLine($"Dict has {dict.Count} keys.");
```

Operator true/false

- `if (x)` *will* call `operator true` on `x` when available
  - So will `while (x)`, `for (…; x; …)` etc.
- `if (!x)` will *not* call operator false
  - Will not compile in our example
- Need `operator!` to allow `if (!x)`

Double Operator Overload

- Two operators can follow one another
  - Operator -->
- Example: preprocess string *then* compare
  - -- can trim and uppercase (reminder: uppercase comparisons are optimized)
  - > can operate as case-insensitive ==

Double Operator Overload Implementation

```
extension(string)
{
  public static string operator --(string s)
    => s.Trim().ToUpperInvariant();

  public static bool operator >
    (string left, string right) =>
    left.Equals(right, StringComparison.CurrentCultureIgnoreCase);

  public static bool operator <(string left, string right)
  {
    throw new NotImplementedException();
  }
}
```

Required for symmetry

It Kinda Works…

- ```
  var input = Console.ReadLine();
  if (input --> "exit") return;
  ```
- Problem: we've now defined > on strings to mean ==
- Operators must be public, so this is a global change
- But you can constrain this by only importing where relevant
- Generally unnecessary since both operations can be folded into one (e.g., >> operator)
  ```

Extension Member Visibility

- Extension members can have any visibility except `protected`
  - And variations thereof (e.g., `protected internal`)
  - There is no inheritance
- Typically `public`
- Can be `private`, useful for internal reuse
  - Extensions only relevant in own static class
  - E.g., `string` extension to validate email relevant only to one class
- `internal`, `file`, etc. – controls scope
- Not applicable to operators (must be public)

Extension Member Visibility

- Behind the scenes, extensions compile as static members
  - Extension blocks do not generate extra classes
- Can be invoked directly (also code completion works)
  - Similar to extension methods
- `extension(string)` with `operator +=` can be called as `StringExtensions.op_AdditionAssignment`

Extension Block Ref Duality

- An extension block can optionally take a `ref` parameter
- Receiver must be
  - A value type; or
  - A generic type constrained to struct
- Required for compound assignment on value types

Extension Block Ref Duality

```
extension(int n)
{
  public bool IsOdd => n % 2 != 0;
}

extension(ref int n)
{
  public void Increment() => ++n;
}

int n = 7;
Console.WriteLine(n.IsOdd); // True
n.Increment();
Console.WriteLine(n); // 8
```

Cannot Override Compiler Intrinsics

- ```
  extension(ref int n)
  {
    public void operator ++() => n--;
  }
  ```
- Will compile!
- Will never be invoked
- Extension have lower priority than existing members
- Same for similar situations (e.g. `ToString()` on `object`)
  ```

Extension Members and Duck Typing

- An extension can expose a duck-typing member
-
```
extension(Foo foo)
{
  public FooEnumerator GetEnumerator() => new(foo);
  public struct FooEnumerator
  {
    public bool MoveNext() => …
    public ? Current => …
  }
}
```
- `foreach (var v in myFoo)` works
- Similar for `GetAwaiter()`, `Deconstruct()`, etc.

Summary

- Extension blocks are a more systematic way of extending class functionality
- Methods, properties, operators
- Static and non-static (excl. operators)
- Ref and non-ref
- Support for duck typing

Thank You!

End of Video

- Enjoy C#!
- X @dnesteruk