

CLARQO PRESS



C# 14 in Depth

*Idiomatic, high-performance
.NET 10 for working engineers*

```
// C# 14 · .NET 10
extension(ReadOnlySpan<char> s) {
    public bool IsBlank => s.IsWhiteSpace();
}
```

Marek Ardenholt

INTERMEDIATE → EXPERT

.NET 10

C# 14 IN DEPTH

Idiomatic, High-Performance .NET 10 for Working Engineers

Marek Ardenholt

Published by **Shadow Wolf Apps** under the *Clarqo Press* imprint ·
clarqo.com/books

First edition · 2026 · © 2026 Shadow Wolf Apps. All rights reserved.

This book was written with the assistance of AI. The code examples are provided as-is, for educational use, without warranty of any kind; verify against the official .NET and C# documentation before relying on them in production.

Distributed free of charge.

Introduction: The Modern C# Landscape

Part I — The Modern Type System

- 1 · The Modern Type System: Values, References, and Everything Between
- 2 · Records, Immutability, and Value Semantics
- 3 · Pattern Matching, List Patterns, and Switch Expressions
- 4 · Nullable Reference Types and Flow Analysis
- 5 · Strings, Text, and Interpolation

Part II — Authoring Modern Types

- 6 · Primary Constructors and Collection Expressions
- 7 · The field Keyword and Modern Property Design
- 8 · Extension Members: Extension Everything (C# 14)

Part III — Generics, Math, and Abstraction

- 9 · Generics Deep Dive: Variance, Constraints, and Specialization
- 10 · Generic Math and Static Virtual/Abstract Interface Members
- 11 · Delegates, Lambdas, and Expression Trees
- 12 · Advanced LINQ and Query Composition

Part IV — High-Performance C

- 13 · Span, Memory, ref structs, and ref Safety
- 14 · stackalloc, Inline Arrays, and Fixed-Size Buffers
- 15 · SIMD and Vectorization: Vector, Vector128/256/512
- 16 · Struct Layout, Allocations, and the Garbage Collector

Part V — Concurrency and Asynchrony

- 17 · async/await Internals and the State Machine
- 18 · ValueTask, Pooling, and Async Performance
- 19 · Async Streams, Channels, and Cancellation
- 20 · Threading, the new Lock Type, Interlocked, and Parallelism

Part VI — Metaprogramming and Interop

- 21 · Reflection, Metadata, and Source Generators
- 22 · Function Pointers, P/Invoke, and Native Interop
- 23 · NativeAOT, Trimming, and Fast Startup

Part VII — Libraries, APIs, and Quality

- 24 · Modern Collections: Frozen, Immutable, and Span-Friendly

- 25 · `System.Text.Json` in Depth
- 26 · Dates, Times, and Ambient State
- 27 · Error Handling, Validation, and Resilience
- 28 · Modern API and Library Design
- 29 · Testing, Benchmarking, and Performance Validation

Part VIII — Reference

- 30 · Capstone Reference: What's New in C# 13 and C# 14
- Appendix · Tooling, Diagnostics, and the Modern Dev Environment

Introduction: The Modern C# Landscape

C# stopped being a single-version language a long time ago. What you actually ship is a *negotiation* between an SDK, a target framework, and a language version — three knobs that fail in confusingly similar ways when they disagree. This chapter calibrates those knobs, sets the measurement discipline the rest of the book leans on, and establishes the version-attribution habit that keeps you out of the “why doesn’t this compile on the build server” trap.

Who this book is for

This is a book for people who already write C# for a living and want the part underneath. We assume you are fluent with generics, `async / await`, LINQ, the type system’s reference/value split, nullable reference types as a *concept* (even if your codebase hasn’t fully adopted them), and that you can read a stack trace without flinching. We do not re-teach `for` loops. We *do* re-teach things you think you know — like what a `record` actually lowers to, why `Span<T>` can’t live on the heap, and when a primary constructor captures a parameter into hidden state versus when it doesn’t.

The throughline is **performance-first but readable**. Those two goals are usually presented as a tradeoff; most of the time they aren’t. The fast path in modern .NET is frequently *also* the idiomatic one, because the BCL and the JIT have been co-designed for a decade to make the obvious code allocate less. Where they genuinely conflict — and they do, around `Span<T>`, `ref struct`, stack allocation, and aggressive inlining — we measure, show the IL or the JIT asm, and then make a deliberate call. We never reach for `unsafe` or a hand-rolled struct enumerator as a reflex. We reach for it with a benchmark in hand.

The release cadence: language, runtime, SDK

Three version numbers move on the same annual November cadence but mean different things. Conflating them is the single most common source of “feature not available” confusion, so internalize the mapping now:

Released	.NET runtime	SDK band	C# language	Support
Nov 2023	.NET 8	8.0.x	C# 12	LTS
Nov 2024	.NET 9	9.0.x	C# 13	STS
Nov 2025	.NET 10	10.0.x	C# 14	LTS

The **runtime** (`net10.0`) determines which BCL APIs exist and which JIT/GC you get. The **language version** (C# 14) determines which *syntax* the compiler accepts. These are independent. You can compile C# 14 syntax that targets `net8.0` for many features — anything that’s pure syntactic sugar lowered by Roslyn (primary constructors, collection expressions, the `field` keyword) needs only a new enough *compiler*, not a new runtime. Other features need runtime or BCL support and will fail or silently degrade on older targets: first-class `Span<T>` conversions, `params Span<T>`, and `System.Threading.Lock` all want types and JIT behavior that older runtimes don’t have. The SDK is what *bundles* a particular Roslyn and a particular default language version; installing the .NET 10 SDK is necessary but, as we’ll see, not sufficient to *get* C# 14.

A compressed feature map for the three versions this book treats as “modern,” because we attribute every feature to its version throughout:

- **C# 12 (.NET 8):** primary constructors on all classes/structs; collection expressions `[a, b, ..spread]`; `ref readonly` parameters; default lambda parameter values; `using` aliases for *any* type (tuples, arrays, generics); inline arrays via `[InlineArray]`.
- **C# 13 (.NET 9):** `params` collections (`params Span<T>`, `params IEnumerable<T>`, any collection type); the dedicated `System.Threading.Lock` type and its `lock` pattern; the

`\e` escape for ESC (`0x1B`); `ref` locals and `unsafe` blocks permitted inside iterators and `async` methods; the `allows ref struct` anti-constraint and `ref struct` interface implementation; implicit index (`^`) in object initializers; `[OverloadResolutionPriority]` ; partial properties and indexers.

- **C# 14 (.NET 10):** the `field` contextual keyword (compiler-synthesized backing field without an explicit one); extension members via `extension` blocks (extension properties, static extension members, extension operators); null-conditional assignment (`x?.Prop = v` , `x?[i] = v`); `nameof` with unbound generics (`nameof(List<>)`); first-class implicit `Span<T>` / `ReadOnlySpan<T>` conversions; partial constructors and partial events; user-defined compound assignment operators; modifiers (`ref` / `out` / `in` / `scoped`) on *implicitly typed* lambda parameters.

How version-attribution callouts work in this book

Every time we introduce syntax, we name the version that shipped it, in prose, like this: “Introduced in C# 14, the `field` keyword...”. We do this because intermediate-to-expert readers don’t read a book front-to-back; they jump to the chapter on collections or `async` and copy a snippet. Without the attribution, that snippet becomes a support ticket when it lands in a `net8.0` library with `<LangVersion>8</LangVersion>` . Treat the version tag as part of the API contract of the snippet — as load-bearing as the type signature.

Feature gating in the csproj: the only screen that matters

Here is a complete, modern project file. Read it as the canonical setup for every sample in this book.

```

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <LangVersion>14</LangVersion>

    <!-- Nullable reference types on, warnings-as-errors for null-safety -->
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>

    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>

    <!-- AOT-ready posture: surfaces trim/AOT incompatibilities at build time -->
    <PublishAot>true</PublishAot>
    <IsTrimmable>true</IsTrimmable>

    <InvariantGlobalization>true</InvariantGlobalization>

    <!-- Make the JIT's choices observable for the disassembly chapters -->

    <ServerGarbageCollection>>false</ServerGarbageCollection>
  </PropertyGroup>

</Project>

```

Two of these lines decide whether your C# 14 code compiles at all, and they are the *first* place to look when a feature “isn’t available”:

`<TargetFramework>net10.0</TargetFramework>` selects the runtime/BCL and, critically, sets the *default* language version. With `net10.0`, the default `LangVersion` is already `14`. So why pin it explicitly? Because the default is a function of the TFM, and the moment someone multi-targets (`<TargetFrameworks>net8.0;net10.0</TargetFrameworks>`) the `net8.0` leg silently drops to C# 12. Pinning `<LangVersion>14</LangVersion>` forces C# 14 syntax across *all* target legs — which then fails fast on `net8.0` for any feature

that needs runtime support, which is exactly the signal you want at build time rather than at runtime.

`<Nullable>enable</Nullable>` (a project-wide knob since .NET 5, but the default in every template since .NET 6) turns nullable reference type analysis on. We keep it on everywhere and pair it with

`<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` so a CS8602 dereference-of-a-possibly-null-reference is a build break, not a warning you'll learn to ignore.

`<PublishAot>true</PublishAot>` doesn't change `dotnet build`, but it makes `dotnet publish` use Native AOT and — more usefully day-to-day — it makes the analyzers flag trim-unsafe and reflection-heavy patterns *as you write them*. Adopting an AOT-ready posture early is cheap; retrofitting it onto a reflection-soaked codebase is not.

Pitfall: the SDK is installed but the language is pinned old

The most expensive five minutes you'll lose this year: you `dotnet --version`, see `10.0.100`, write an `extension` block, and the compiler says *"Feature 'extension members' is not available in C# 12. Please use language version 14.0 or greater."* The SDK is fine. The *project* pinned

`<LangVersion>12</LangVersion>` (or inherited it from a multi-target leg, or a `Directory.Build.props` set it org-wide). The compiler error text names the version — read it. The fix is in the `csproj`, never in your SDK install.

Pitfall: confusing runtime version with language version

The mirror-image failure. You target `net8.0`, set `<LangVersion>14</LangVersion>`, and a `params ReadOnlySpan<T>` overload either won't resolve to the span form or behaves like the old `params T[]`. That's not a language problem — C# 14 the *syntax* is available — it's a *runtime/BCL* problem: the span-based overloads and the JIT support they rely on live in .NET 9+. "Language version" answers *can the compiler parse this?*; "target framework" answers *does the runtime have*

the type and codegen to back it?. When a feature half-works, ask which of those two you're actually missing before you touch anything.

A minimal Program.cs that sets the tone

Top-level statements (C# 9) and file-scoped namespaces (C# 10) are old enough that we treat them as the baseline, not a feature. Collection expressions (C# 12) replace the `new[] {...} / new List<int> {...}` ceremony with a target-typed `[...]` that the compiler lowers to the cheapest construction for the destination type — a literal array for `T[]`; a `List<T>` built in place via `CollectionsMarshal.SetCount + AsSpan` (one allocation, no resize churn) when the length is known; the builder for an `ImmutableArray<T>`; and, for a `Span<T> / ReadOnlySpan<T>` target, a `stackalloc` or inline-array buffer with no heap allocation at all.

```
// Program.cs – top-level statements, no Main, no
// enclosing class.
using System.Globalization;

// Collection expression (C# 12). Target-typed to
// int[]: lowered to a literal array.
int[] readings = [21, 19, 23, 18, 25, 22];

// Spread (..) composes collections without an
// intermediate allocation per element.
int[] withSentinels = [0, ..readings, 0];

double mean = readings.Average();
int peak = readings.Max();

Console.WriteLine(string.Create(CultureInfo.InvariantCulture,
    $"n={readings.Length} mean={mean:F1} peak={peak}
    padded={withSentinels.Length}"));
// n=6 mean=21.3 peak=25 padded=8
```

There is no class here, no `static void Main`. The compiler synthesizes a `Program` class with a `Main` for you; `args` is in scope implicitly if you reference it. This particular `string.Create` overload — `Create(IFormatProvider?, ref`

`DefaultInterpolatedStringHandler`) — is chosen because it lets you pass an explicit `CultureInfo.InvariantCulture` and route through the interpolated-string-handler machinery (C# 10) in one call. The handler's `AppendFormatted<T>(T value, ...)` is generic, so each `int` / `double` is formatted without the `object` -boxing that the old `string.Format(IFormatProvider, string, object?[])` path incurs for value-type arguments — a small thing that matters in hot paths and that we'll return to.

Global usings and aliasing any type

`<ImplicitUsings>enable</ImplicitUsings>` injects a curated set of `global using` directives (`System`, `System.Linq`, `System.Collections.Generic`, ...). You add your own in one file and they apply solution-wide. Introduced in C# 12, the `using` alias can now name *any* type — including tuples, arrays, and constructed generics — not just named types. This is the cleanest way to give a structurally-typed value a name without paying for a wrapper struct:

```
// GlobalUsings.cs – one file, applies to the whole
// project.
global using System.Diagnostics;
global using System.Runtime.CompilerServices;

// C# 12: alias an arbitrary type. This names a tuple
// shape; it is NOT a new type.
global using GeoPoint = (double Lat, double Lon);

// C# 12: alias a constructed generic, too.
global using StringMap =
    System.Collections.Generic.Dictionary<string,
    string>;
```

```

// Usage anywhere in the project – GeoPoint is
// structurally a (double, double).
GeoPoint prague = (50.0755, 14.4378);
GeoPoint brno   = ParseCity("Brno");

static GeoPoint ParseCity(string _) => (49.1951,
    16.6068);

// Because it's an alias, not a wrapper, this
// assignment is legal across the seam:
(double lat, double lon) raw = prague; // no
// conversion, same underlying ValueTuple

```

The tradeoff is real and worth stating: an alias is *not* a nominal type. `GeoPoint` and `(double, double)` are interchangeable, which means you get zero type-safety against mixing up a `GeoPoint` and a `(double Width, double Height)`. When you want the compiler to *stop* you from passing latitude where you meant width, reach for a `readonly record struct`, not an alias. The alias buys readability, not safety.

The measurement template: BenchmarkDotNet

Every performance claim in this book is backed by a benchmark, and they all follow one shape. `BenchmarkDotNet` (`BenchmarkDotNet` NuGet, run from a Release build) handles warmup, statistical iteration, and — via `MemoryDiagnoser` — allocation accounting. This is the harness you'll see, verbatim in spirit, for the rest of the book:


```
// Benchmarks.cs – the book's measurement template.
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

BenchmarkRunner.Run<SumBench>();

[MemoryDiagnoser] //
    reports Allocated bytes/op + GC counts
public class SumBench
{
    private readonly int[] _data = [..
        Enumerable.Range(0, 1024)];

    [Benchmark(Baseline = true)]
    public long Linq() => _data.Sum(); //
        delegate + enumerator overhead

    [Benchmark]
    public long Loop() //
        zero-allocation, JIT-friendly
    {
        long sum = 0;
        foreach (int x in _data) sum += x;
        return sum;
    }
}
```

Run it with `dotnet run -c Release`. Two columns earn their keep: **Ratio** (relative to the `Baseline = true` method) and **Allocated** (bytes per operation). When `Allocated` reads `-`, the method allocated nothing — the bar we hold the hot path to. `[MemoryDiagnoser]` works by reading `GC.GetAllocatedBytesForCurrentThread()` and the per-generation collection counts around the measured runs, so it attributes managed allocations to each benchmark essentially for free; it's the cheapest insurance against accidental allocation regressions, which is why it's on every benchmark in this book. Never benchmark a Debug build: without optimizations the JIT skips inlining, omits enregistration, and inserts debug-only bookkeeping, so the numbers become fiction. BenchmarkDotNet refuses to run an assembly built in Debug and will tell you so.

Conventions: SharpLab, IL, and the disassembly mindset

We inspect lowering constantly. Three tools recur:

- **SharpLab** (sharplab.io) — paste C#, see the Roslyn-lowered C#, the IL, or the JIT asm, for a chosen language version. This is how we *prove* that a primary constructor captured a parameter into a field, or that a collection expression became a `stackalloc`. When a chapter says “this lowers to,” that claim was checked in SharpLab.
- **IL inspection** — `ildasm`, ILSpy, or `dotnet-ildasm`. IL tells you what the compiler decided; asm tells you what the JIT decided. We read IL to settle questions about boxing, defensive copies of `readonly struct`s, and call-vs-callvirt dispatch.
- **The disassembly mindset** — when behavior surprises you, drop a level. A `record`’s `Equals` is generated IL you can read. A `Span<T>` indexer is a couple of asm instructions you can count. “Surprising” almost always means “I had the wrong mental model of the lowering,” and the lowering is *right there*.

You don’t need to memorize IL opcodes. You need the reflex to check, and a rough sense of what’s expensive: a `newobj` in a loop is a heap allocation; a `box` is an allocation plus a copy; a `callvirt` on a sealed type is a missed devirtualization opportunity.

The running example domains

Rather than disconnected toy snippets, the book reuses a few small domains so you build intuition across chapters:

- **Telemetry ingestion** — fixed-size sensor readings flowing through `Span<T>` pipelines. The vehicle for stack allocation, `ref struct`, first-class spans, and zero-allocation parsing.
- **An order/pricing domain** — `record` and `readonly record struct` value objects (`Money`, `Sku`, `OrderLine`).

The vehicle for value semantics, `with` expressions, pattern matching, and primary constructors.

- **A tiny in-memory event log** — concurrent appends and reads. The vehicle for the C# 13 `Lock` type, `Channel<T>`, and the `async/threading` chapters.

Worked example: the same idea, three C# eras

To tie the chapter together — and to make the version-attribution habit concrete — here is one small value object evolving across the three modern language versions. It computes a discounted line total and exposes a validated, mutable label.

```

using System.Globalization;

// readonly record struct (C# 10): value semantics, no
// heap allocation, no defensive copies.
// The positional parameters are a record's primary
// constructor – records (incl. record
// structs) have had these since C# 9/10; C# 12 only
// extended primary constructors to plain
// (non-record) classes and structs, as on OrderLine
// below.
public readonly record struct Money(decimal Amount,
    string Currency)
{
    public static Money operator *(Money m, decimal
        factor) => m with { Amount = m.Amount * factor
    };

    public override string ToString() =>
        string.Create(CultureInfo.InvariantCulture, $"
            {Amount:0.00} {Currency}");
}

public sealed class OrderLine(Sku sku, int quantity,
    Money unitPrice)
{
    // C# 14: the `field` keyword. A validated
    // property with NO hand-written backing field.
    // Replaces the old private-field-plus-two-
    // accessors boilerplate.
    public string Label
    {
        get;
        set => field =
            string.IsNullOrEmpty(value)
                ? throw new ArgumentException("Label
                    required", nameof(value))
                : value.Trim();
    } = sku.Code; // primary-ctor param (C# 12)
    // in the initializer. NB: a property
    // initializer writes the
    // backing field DIRECTLY – it does NOT run
    // the validating setter, so
    // sku.Code lands untrimmed/unvalidated.

    public int Quantity { get; } = quantity > 0
        ? quantity
        : throw new
            ArgumentOutOfRangeException(nameof(quantity));

    // Ordinary optional parameter (a C# 1-era
    // feature) – NOT to be confused with the
    // C# 12 "default lambda parameter values"; this
    // is a method, not a lambda.

```

```
    public Money Total(decimal discount = 0m) =>
        unitPrice * Quantity * (1 - discount);
}
```

```
public readonly record struct Sku(string Code);
```

```
// Extension members (C# 14): an `extension` block,
    not the old `this`-parameter syntax.
// Static class, non-generic, non-nested – the
    compiler requirement for extension blocks.
```

```
public static class OrderLineExtensions
{
    extension(OrderLine line)
    {
        // Extension PROPERTY (C# 14) – impossible
            before; extension methods couldn't be
            properties.
        public bool IsBulk => line.Quantity >= 100;

        // Extension method, now living in the same
            receiver block.
        public Money DiscountedTotal() =>
            line.Total(line.IsBulk ? 0.15m : 0m);
    }
}
```

```
// Driving code.
var line = new OrderLine(new Sku("WIDGET-9"),
    quantity: 120, unitPrice: new Money(4.50m,
        "EUR"))
{
    Label = " Premium Widget " //
        trimmed + validated by the `field` setter
};

Console.WriteLine(line.Label);
// Premium Widget
Console.WriteLine(line.IsBulk);
// True (extension property, C# 14)
Console.WriteLine(line.DiscountedTotal());
// 459.00 EUR

// Null-conditional assignment (C# 14): assign through
// a possibly-null reference, no if-guard.
OrderLine? maybe = Lookup("WIDGET-9");
maybe?.Label = "Renamed";
// RHS evaluated only if maybe is non-null

static OrderLine? Lookup(string _) => null;
// returns null here → assignment is a no-op
```

What this demonstrates, with versions attributed:

- The `readonly record struct Money` (C# 10) gives value equality and immutability with **zero heap allocation** — `Money` lives on the stack or inline in its container. Marking it `readonly` lets the *compiler* prove no member mutates `this`, which eliminates the *defensive copies* Roslyn would otherwise emit whenever you call an instance member through a read-only reference to the struct — a `readonly` field, an `in` parameter, or a `static readonly`. Those copies are an IL-level artifact (`ldobj` / `stobj` or a stack temporary), not a JIT decision; verify in SharpLab that with `readonly` the copy dance is simply absent.
- The `field` keyword (C# 14) collapses the validated-property boilerplate. Before C# 14 the `Label` property needed an explicit `private string _label;` and both accessors referencing it. The lowering is *identical* — Roslyn still synthesizes a backing field — so there's no runtime cost, only less code to get wrong. Watch the one gotcha the docs call out: because

`field` is now a contextual keyword, if your type already has a member literally named `field`, then inside a property accessor body an unqualified `field` binds to the synthesized backing field, not your member — a silent binding change from C# 13. The compiler raises a warning at language version 14, and the fix is to escape the identifier as `@field` (or qualify an instance member as `this.field`).

- The `extension(OrderLine line)` block (C# 14) lets `IsBulk` be a *property*, which the classic `static bool IsBulk(this OrderLine line)` syntax could never express — extension methods could only ever be methods. The lowering is still a static method; the property is a compiler-level affordance, so there's no dispatch cost beyond a normal static call (and the JIT inlines trivial ones).
- Null-conditional assignment (C# 14) replaces the `if (maybe is not null) maybe.Label = ...` guard with `maybe?.Label = ...`, and — this is the part people miss — the right-hand side is **not evaluated** when the receiver is null. If the RHS were an expensive `GetCurrentOrder()` call, you'd skip it for free. Increment/decrement (`maybe?.Count++`) remain disallowed; assignment and compound assignment (`+=`, `--`) are in.

When *not* to reach for these

Modern syntax is not free of judgment. A few standing rules we'll reinforce throughout:

- A `readonly record struct` is the wrong choice once it grows past roughly the size of a couple of pointers and gets passed around by value a lot — the copies overwhelm the no-allocation win. Measure with the template above; if `Money` were sixteen fields you'd want a `class` or `in`-passing.
- The `field` keyword is great for *validated* or *lazy* properties. Don't use it to dress up a plain auto-property — `public string Name { get; set; }` is already minimal, and adding a `field` body just adds surface area.
- Extension *everything* invites putting domain logic in static classes far from the type. Extension members are for *aug-*

menting types you don't own (BCL types, generated code) and for genuinely cross-cutting helpers — not as a substitute for instance methods on types you control.

- Null-conditional assignment quietly turns “do nothing when null” into the default. That’s occasionally a bug waiting to happen: if a null receiver *should* be an error, the explicit `if /throw` is clearer than a silent no-op.

With the project file pinned, the benchmark harness ready, and the disassembly tools at hand, you’re equipped to read the rest of this book the way it’s written — every claim measurable, every feature dated, every fast path checked against the IL.

Part I — The Modern Type System

1 · The Modern Type System: Values, References, and Everything Between

The .NET type system is often taught as a binary split — “classes go on the heap, structs go on the stack” — and that folklore is wrong in both directions. The truth is a three-way model (reference types, value types, and the byref-like `ref struct` family) layered over a runtime that unifies generics, boxes value types on demand, and treats nullability as two entirely separate, unrelated mechanisms. This chapter rebuilds that mental model from the metal up, attributing each modern affordance to the C# version that introduced it, and shows where the abstractions leak in ways that cost you allocations, defensive copies, or correctness.

One model, three categories

Every type in C# falls into exactly one of three storage-and-identity categories, and almost every confusing C# behavior dissolves once you internalize which one you are holding.

A **reference type** (`class` , `interface` , `delegate` , array, `string` , `record class`) is always allocated on the GC heap. A variable of a reference type holds a *reference* — an obscured machine pointer — to that heap object. Copying the variable copies the reference, not the object; two variables can observe each other’s mutations. Identity is the object’s address: two references are `ReferenceEquals` iff they point at the same allocation.

A **value type** (`struct` , `enum` , primitives, `record struct` , `ValueTuple` , `Nullable<T>`) has no separate object header and no inherent heap home. Its storage is *inlined* wherever it is declared: a local lives in a register or stack slot, a field lives inside its containing object, an array element lives inside the array’s payload. Copying the variable copies the *bits*. There is no identity beyond value equality — two `int` s holding `5` are indistinguishable.

The categorical line is **not** “stack vs heap.” A value type lives on the heap whenever its container does: an `int` field inside a `class` is on the heap; a `Point[]` puts every `Point` on the heap inline. The real distinction is *copy semantics and identity*, not physical location. A reference type’s instance is always heap-allocated; a value type’s storage follows its container.

The third category — `ref struct` (byref-like types such as `Span<T>` and `ReadOnlySpan<T>`) — is a value type with an extra rule enforced by the compiler and verified by the runtime: an instance may never reside on the GC heap. We return to it shortly, because it is where C# 13 and C# 14 spent significant design effort.

```
struct PointVal { public int X, Y; }
class PointRef { public int X, Y; }

var a = new PointVal { X = 1 };
var b = a;           // copies 8 bytes
b.X = 99;
// a.X == 1, b.X == 99 – independent

var c = new PointRef { X = 1 };
var d = c;           // copies a reference
d.X = 99;
// c.X == 99 – same object observed twice
```

`readonly struct`, `readonly` members, and the defensive copy tax

A mutable struct exposed through a `readonly` field or an `in` parameter triggers one of the most expensive silent behaviors in C#: the **defensive copy**. When the compiler cannot prove a method leaves a struct unmutated, and the struct is reached through a read-only location, it copies the entire struct before invoking the method — every property `get`, every method call — to protect the read-only guarantee. For a large struct in a hot loop this is a per-call `memcpy` you never wrote.

The cure (C# 7.2, but central to modern idiom) is `readonly struct`: it makes *every* instance member implicitly `readonly`,

so the compiler knows no member can mutate `this`, and the defensive copy vanishes. You can also mark individual members `readonly` on an otherwise-mutable struct.

Here is the canonical `Money` value type — `readonly struct`, value-equal, and consumed through `in` with zero defensive copies:

```

using System.Globalization;

public readonly struct Money : IEquatable<Money>,
    IComparable<Money>
{
    public decimal Amount { get; }
    public string Currency { get; }

    public Money(decimal amount, string currency)
        => (Amount, Currency) = (amount, currency);

    // Implicitly readonly because the struct is
    readonly: no copy on access.
    public Money Add(in Money other)
    {
        if (Currency != other.Currency)
            throw new InvalidOperationException(
                $"Cannot add {Currency} to {other.Currency}.");
        return new Money(Amount + other.Amount,
            Currency);
    }

    public int CompareTo(Money other) =>
        Amount.CompareTo(other.Amount);
    public bool Equals(Money other)
        => Amount == other.Amount && Currency ==
            other.Currency;
    public override bool Equals(object? obj) => obj is
        Money m && Equals(m);
    public override int GetHashCode() =>
        HashCode.Combine(Amount, Currency);

    public override string ToString()
        => Amount.ToString("F2",
            CultureInfo.InvariantCulture) + " " +
            Currency;
}

static Money SumLedger(in Money opening,
    ReadOnlySpan<Money> entries)
{
    var total = opening; // one copy,
    deliberate
    foreach (ref readonly var e in entries)
        total = total.Add(in e); // no
    defensive copy: Money is readonly
    return total;
}

```

Because `Money` is a `readonly struct`, `opening.Add(...)` on the `in` parameter and `total.Add(in e)` over `ref readonly` iteration variables compile to direct calls on the original storage. Had `Money` been a plain `struct`, each `e.Add(...)` through `ref readonly / in` would have copied the full 24 bytes of `Money` (a 16-byte `decimal` plus an 8-byte string reference on a 64-bit runtime) first. Confirm it yourself in a disassembler: the `readonly struct` version shows no hidden `Money` copy at the call site; drop the `readonly` and a `stobj` /copy materializes before the call.

Pitfall — mutating through a `readonly` field operates on a copy. Expose a *mutable* struct via a `readonly` field and call a mutating method, and the compiler silently copies the field first; your mutation lands on the throwaway copy and the field is unchanged. The original `Counter` keeps its value, no warning emitted:

```
struct Counter { public int N; public void Inc() =>
    N++; }

class Holder
{
    public readonly Counter Field = new();
    public void Bump() => Field.Inc(); // mutates a
    COPY; Field.N stays 0
}
```

The fix is the same triad: make the struct `readonly` (so `Inc` can't exist as written — surfacing the bug at compile time), or don't store mutable structs in `readonly` fields. This class of bug is exactly why mutable structs are an anti-pattern; prefer `readonly struct` with `with`-style copy methods.

The `ref struct` family: why it cannot touch the heap

`Span<T>` and `ReadOnlySpan<T>` are `ref struct`s. A `ref struct` may contain a *managed pointer* (a `ref` field — `ref T` — formalized in C# 11) into the interior of an array, a stack

frame, or unmanaged memory. The GC tracks object references but cannot relocate or even reason about an interior pointer that has escaped to the heap, so the runtime forbids a `ref struct` from ever living there. That single rule cascades into a long list of compiler restrictions: a `ref struct` cannot be a field of a class or non-`ref struct`, cannot be boxed, cannot be a generic type argument (with the C# 13 `allows ref struct` opt-in escape, below), and historically could not appear in `async` methods, iterators, or lambdas that capture it.

C# 13 (.NET 9) relaxed two of those walls. `ref` locals and `unsafe` contexts are now permitted in iterators and `async` methods (you still cannot keep a `ref struct` across an `await` / `yield`, but you may use one between suspension points). And the new `allows ref struct` anti-constraint lets a generic type parameter accept `ref struct` arguments, so generic algorithms can flow `Span<T>` through without boxing:

```
// C# 13: T may be a ref struct (e.g.
//      ReadOnlySpan<char>) because of 'allows ref
//      struct'.
// The anti-constraint must come LAST in the where
//      clause, after every real constraint.
static int CountWhere<T>(T span, ReadOnlySpan<char>
    needle)
    where T : IEnumerable<char>, allows ref struct
{
    int count = 0;
    foreach (char c in span)           // requires T :
        IEnumerable<char>
            if (needle.Contains(c)) count++;
    return count;
}
```

The constraint ordering is load-bearing: the compiler requires `allows ref struct` to be the final item in a type parameter's constraint list, and it cannot be combined with `class` / `class?`. In real BCL code the more common shape is a bare `allows ref struct` with no interface constraint (e.g. `MemoryExtensions` helpers that accept `Span<T>` or array through one generic parameter); the `IEnumerable<char>` pairing above just shows that the anti-constraint composes with ordinary constraints.

Here is a real stack-only abstraction — a forward-only CSV field reader over a `ReadOnlySpan<char>`. It allocates nothing, never copies the underlying buffer, and the compiler structurally prevents it from outliving its source:


```

public ref struct CsvLineReader
{
    private ReadOnlySpan<char> _remaining;
    private readonly char _delimiter;

    public CsvLineReader(ReadOnlySpan<char> line, char
        delimiter = ',')
    {
        _remaining = line;
        _delimiter = delimiter;
    }

    // Returns the next field as a slice of the
    // original buffer – zero allocation.
    public bool TryReadField(out ReadOnlySpan<char>
        field)
    {
        if (_remaining.IsEmpty) { field = default;
            return false; }

        int idx = _remaining.IndexOf(_delimiter);
        if (idx < 0)
        {
            field = _remaining;
            _remaining = default;
        }
        else
        {
            field = _remaining[..idx];
            _remaining = _remaining[(idx + 1)..];
        }
        return true;
    }
}

static decimal SumThirdColumn(ReadOnlySpan<char>
    csvLine)
{
    var reader = new CsvLineReader(csvLine);
    int col = 0;
    while (reader.TryReadField(out var field))
    {
        if (col++ == 2)
            return decimal.Parse(field, provider:
                CultureInfo.InvariantCulture);
    }
    return 0m;
}

// SumThirdColumn("a,b,42.50,c") == 42.50m, with no
// string allocations.

```

`CsvLineReader` cannot be stored in a list, captured by a lambda, or returned from an `async` method — and that is the point. The type is the lifetime contract.

C# 14 makes spans first-class. Before .NET 10 you constantly wrote `arr.AsSpan()` or relied on a `T[] → ReadOnlySpan<T>` conversion that didn't compose with user-defined conversions or generic inference. C# 14 adds **first-class implicit conversions** between `T[]`, `Span<T>`, and `ReadOnlySpan<T>`. These are *free* — they reinterpret an existing buffer in a span wrapper, never copy data, and never allocate — and they now compose with extension-method receivers and generic type inference:

```
// C# 14: implicit array → ReadOnlySpan<char>, no
// .AsSpan() needed.
char[] buffer = { 'a', ',', 'b', ',', '7' };
var sum = SumThirdColumn(buffer); // implicit char[]
// → ReadOnlySpan<char>
```

The conversions are intentionally direction-safe: you still cannot implicitly turn a `ReadOnlySpan<T>` into a writable `Span<T>`, so read-only intent survives.

Two nullabilities that share nothing

`int?` and `string?` look symmetric. They are not even the same *kind* of feature, and conflating them is a durable source of bugs.

`Nullable<T>` (**T? for value types**) is a real `readonly struct` in the BCL — `System.Nullable<T>` — introduced in C# 2. It carries a `bool HasValue` and a `T value`. It is a runtime construct with its own boxing rules: boxing a `Nullable<T>` with `HasValue == false` boxes to a plain `null` reference; boxing one with a value boxes the *underlying* `T`, not a `Nullable<T>`. There is genuine machinery here — IL opcodes, a distinct type, lifted operators.

Nullable reference annotations (`string?`, NRTs) are, by contrast, *pure compile-time metadata* introduced in C# 8. At runtime `string?` and `string` are the *same type* — `System.String`. There is no `Nullable<string>`, no

`HasValue`, no wrapper. The `?` only feeds the compiler's flow analysis and emits `[Nullable]` / `[NullableContext]` attributes so other assemblies see your intent. Disable `<Nullable>` or ignore the warnings and the annotations evaporate; the runtime never enforces them.

```
int? maybe = null;
Console.WriteLine(maybe is null);           // True
Console.WriteLine(maybe.GetType());         // throws
                                           NRE – there's no boxed value to ask!

int? present = 5;
Console.WriteLine(present.GetType());       //
                                           System.Int32 – NOT Nullable<int>; boxing
                                           unwrapped it

string? s = null;
Console.WriteLine(typeof(string?) == typeof(string));
                                           // True – same runtime type
```

Pitfall. Do not assume the two share machinery. You can write `int?` in a generic `where T : struct` context and rely on `Nullable<T>` semantics; you *cannot* get any runtime guarantee from `string?`. A `Dictionary<string, int?>` has nullable *values* with real `HasValue` checks; a `Dictionary<string, string?>` is just a `Dictionary<string, string>` that the compiler nags you about. And `default(int?)` is a struct with `HasValue == false`, whereas `default(string?)` is a literal `null` reference — same spelling, different worlds.

Tuples, `ValueTuple`, deconstruction, and naming rules

The `(int, string)` tuple syntax (C# 7) lowers to `System.ValueTuple<...>`, a mutable `struct`. Because it is a value type, tuples are copied by value and allocate nothing on the heap. The crucial subtlety: **element names are compile-time-only metadata**, not part of the runtime type. `(int Id, string Name)` and `(int A, string B)` are the *same* `ValueTuple<int, string>` at runtime; the names are erased and only the compiler knows them (it surfaces them through

[`TupleElementNames`] for cross-assembly use). This is why two tuples with different element names are assignable to each other and compare equal by position.

```
(int Id, string Name) user = (1, "Ada");
(int A, string B) other = user;    // legal: same
                                   ValueTuple<int,string>
Console.WriteLine(user == (1, "Ada")); // True -
                                   positional value equality

// Deconstruction
var (id, name) = user;
var (_, onlyName) = user;          // discard the
                                   first element
```

Deconstruction is structural, not tuple-specific: any type with a `Deconstruct` method (or extension) participates. Combined with positional patterns and discards (`_`), `Deconstruct` lets your own types flow through the same syntax:

```
readonly struct GeoPoint(double lat, double lon)
{
    public double Lat { get; } = lat;
    public double Lon { get; } = lon;
    public void Deconstruct(out double lat, out double
        lon)
        => (lat, lon) = (Lat, Lon);
}

static string Classify(GeoPoint p) => p switch
{
    (0, 0)                => "null island",
    (> 0, _)              => "northern hemisphere",
    (var lat, _) when lat < 0 => "southern
        hemisphere",
    -                      => "equator",
};

var (lat, _) = new GeoPoint(51.5, -0.12); // discard
        longitude
```

The positional patterns `(0, 0)` and `(> 0, _)` invoke `Deconstruct` under the hood, then match each position; `_` discards. Note the constructor uses a **primary constructor** (C#

12 extended primary constructors to all structs and classes), and the parameters initialize the auto-properties directly.

`using` aliases for any type (C# 12)

Before C# 12, `using` aliases were limited to named types. C# 12 lifted that to **any type**, including tuples, arrays, pointers, and generics. The headline use is naming a tuple shape once and sharing it across methods, which restores readability that positional tuples otherwise erode:

```
using Point = (double X, double Y);           // C# 12:
        alias a tuple type
using Matrix = double[,];                     // alias
        an array type
using Handle = nint;                           // alias
        a native-int

static double Distance(Point a, Point b)
{
    double dx = a.X - b.X, dy = a.Y - b.Y;
    return Math.Sqrt(dx * dx + dy * dy);
}

static Point Midpoint(Point a, Point b) => ((a.X +
    b.X) / 2, (a.Y + b.Y) / 2);

// Consumed across methods with the names intact:
Point origin = (0, 0), target = (3, 4);
double d = Distance(origin, Midpoint(origin, target));
        // 2.5
```

The alias is purely a source-level convenience — `Point` is `ValueTuple<double, double>` to the runtime, so the element names `X / Y` come from the alias declaration and are visible wherever the alias is in scope. Aliasing keeps the call sites self-documenting without introducing a wrapper type or any allocation.

`default`, `default(T)`, `null`, and `zero`

`default(T)` (and the target-typed `default`, C# 7.1) yields the type's zero value: all-bits-zero for value types, `null` for refer-

ence types. The trap is treating `default` as interchangeable with `null` or `0`.

```
Console.WriteLine(default(int));           // 0
Console.WriteLine(default(int?) is null); // True -
    Nullable<int> with HasValue=false
Console.WriteLine(default(string) is null); // True
Console.WriteLine(default(DateTime) ==
    DateTime.MinValue); // True - 0001-01-01
    00:00:00, NOT null, NOT a sentinel
Console.WriteLine(default(Money));         // "0.00 " -
    trailing space then nothing: Currency is null!
```

`default(Money)` is the sharp edge: it bypasses your constructor entirely, producing a `Money` whose `Currency` is `null` even though your only constructor forbids that. Every `struct` is implicitly default-constructible to all-zeros, and there is no way to prevent it. This is why a `readonly struct` must treat the zero state as valid (or detect and reject it at use sites); you cannot rely on a constructor invariant for value types. In generics, `default(T)` is the only safe “empty” expression that works for both `class T` (gives `null`) and `struct T` (gives zero) — which is precisely why the BCL uses it instead of `null`.

Boxing: where it still hides

Boxing wraps a value type in a heap object so it can be referenced as `object` or an interface. It costs an allocation, a copy, and eventual GC pressure, and modern C# has eliminated *many* but not *all* of its silent occurrences. The two that still bite:

1. Using a struct through an interface it implements.

Assigning a struct to an interface-typed variable boxes it. This is the most common accidental box because it looks like ordinary polymorphism.

```

struct Tick : IComparable<Tick>
{
    public long Ns;
    public int CompareTo(Tick other) =>
        Ns.CompareTo(other.Ns);
}

// Parameter typed as the interface – every Tick
// argument is boxed at the call site.
static int CompareViaInterface(IComparable<Tick> left,
    Tick right)
    => left.CompareTo(right);

var t = new Tick { Ns = 100 };
int r = CompareViaInterface(t, new Tick { Ns = 50 });
    // boxes `t`: box Tick before the call

```

In a disassembler the call site shows a literal `box Tick` instruction before the argument is passed — a hidden ~24-byte heap allocation per call (a `Tick` is 8 payload bytes plus the boxed object's 16-byte header on a 64-bit runtime). The fix is to take `T` generically and constrain `where T : IComparable<T>`, which dispatches through a *constrained* call **without** boxing the value type:

```

static int CompareUnboxed<T>(T left, T right) where T
    : IComparable<T>
    => left.CompareTo(right); // constrained
    // callvirt – no box for value types

int r = CompareUnboxed(new Tick { Ns = 100 }, new Tick
    { Ns = 50 }); // 1, zero allocation

```

Pitfall — a `struct` implementing an interface, used through that interface, boxes on every call. Generic constraints (`where T : IFoo`) and constrained calls are how you keep value-type dispatch allocation-free. `foreach` over a `List<T>.Enumerator` (a struct) avoids the box only because the compiler pattern-matches the enumerator by structural shape; iterate the same collection through `IEnumerable<T>` and you box the enumerator on every loop.

2. String interpolation and `params object[]`.

`string.Format`-style APIs and any `params object[]` box each value-type argument. The interpolated string handler machinery (C# 10 / .NET 6) dodges this: building a plain `string` from `$"value is {42}"` now lowers to a `DefaultInterpolatedStringHandler` whose generic `AppendFormatted<T>(T)` is specialized per value type, so the `42` is *not* boxed. Some APIs go further and accept the handler directly — `StringBuilder.Append($"...")` binds to a dedicated `AppendInterpolatedStringHandler` overload (.NET 6) and appends each hole straight into the builder with no intermediate string. But `string.Format("{0}", 42)` still boxes the `42`, because its signature is `params object?[]` and the runtime sees an `object`, not the handler. Watch the API surface, not the syntax: an interpolated *literal* avoids the box; an explicit `object[]` / `string.Format` call does not.

Type identity, generic unification, and the runtime

At runtime the CLR maintains one canonical `System.Type` per *constructed* type, and that identity drives generics. For **reference type** arguments the JIT shares a single native code body across all of them — `List<string>` and `List<FileStream>` reuse the same compiled method instantiations because references are all pointer-shaped. For **value type** arguments each distinct instantiation gets its *own* specialized code (`List<int>` and `List<double>` are separate native bodies), which is exactly what makes generics over value types allocation-free and as fast as hand-written code — no boxing, no shared-bloat indirection. This is “generic specialization,” and it is the performance reason to prefer `List<int>` over `ArrayList`.

Type *identity* also explains tuple/alias erasure (names gone, `ValueTuple` survives), NRT erasure (`string? ≡ string`), and `nameof` behavior. C# 14 sharpens the last point: **`nameof` now accepts an unbound generic type**, so `nameof(List<>)` is legal and yields `"List"` — useful in diagnostics, source genera-

tors, and logging where you previously had to supply a dummy type argument:

```
// C# 14: unbound generic in nameof  
string name = nameof(Dictionary<,>); // "Dictionary"
```

A worked example: a low-allocation parser that ties it together

This routine parses a `key=value;...` connection-string fragment over a `ReadOnlySpan<char>`, returns a `readonly struct` result by value, leans on C# 14 implicit span conversions and `field`-backed validation, and allocates nothing on the success path. It exercises `ref struct` scanning, value-type results, deconstruction, and the boxing-avoidance discipline from this chapter.

```

using System;
using System.Globalization;

// C# 14: `field` keyword – compiler-synthesized
//       backing field, no manual field.
public readonly struct PoolConfig
{
    public int MaxSize
    {
        get;
        init => field = value > 0
            ? value
            : throw new
                ArgumentOutOfRangeException(nameof(MaxSize));
    }
    public TimeSpan Timeout { get; init; }

    public void Deconstruct(out int maxSize, out
        TimeSpan timeout)
        => (maxSize, timeout) = (MaxSize, Timeout);
}

public static class ConfigParser
{
    public static PoolConfig Parse(ReadOnlySpan<char>
        text)
    {
        int max = 10;
        var timeout = TimeSpan.FromSeconds(30);

        var rest = text;
        while (!rest.IsEmpty)
        {
            int semi = rest.IndexOf(';');
            ReadOnlySpan<char> pair = semi < 0 ? rest
                : rest[..semi];
            rest = semi < 0 ? default : rest[(semi +
                1)..];

            int eq = pair.IndexOf('=');
            if (eq < 0) continue;

            ReadOnlySpan<char> key =
                pair[..eq].Trim();
            ReadOnlySpan<char> value = pair[(eq +
                1)..].Trim();

            if (key.Equals("max",
                StringComparison.OrdinalIgnoreCase))
                max = int.Parse(value, provider:
                    CultureInfo.InvariantCulture);
        }
    }
}

```

```

        else if (key.Equals("timeout",
StringComparison.OrdinalIgnoreCase))
            timeout = TimeSpan.FromSeconds(
                double.Parse(value, provider:
CultureInfo.InvariantCulture));
        }

        return new PoolConfig { MaxSize = max, Timeout
= timeout };
    }
}

// Usage – string → ReadOnlySpan<char> has been
// implicit since spans shipped;
// char[] → ReadOnlySpan<char> as a first-class
// conversion is the C# 14 addition.
var (size, timeout) = ConfigParser.Parse("max = 50 ;
timeout = 5");
Console.WriteLine($"{size} / {timeout}"); // 50 /
00:00:05

```

Everything load-bearing here is a value: `PoolConfig` is a `readonly struct` returned by value (no heap object), the scanning uses span slices into the original buffer (no substring allocations), and `int.Parse` / `double.Parse` accept `ReadOnlySpan<char>` directly so the per-field text is never materialized as a `string`. The `field` keyword (C# 14) lets `MaxSize` validate in its `init` accessor without a hand-written backing field. Deconstruction yields the result positionally. The only thing that *would* allocate is an exception on bad input — the failure path, not the throughput path.

Performance notes and when not to reach for these tools

Don't make everything a struct. Value types win when they are small (rule of thumb: $\leq$ a few machine words), immutable, and copied rarely or passed by `in` / `ref`. A large mutable struct copied through method calls loses to a class on both speed (repeated copies) and correctness (the copy-mutation pitfalls above). Above roughly 16–24 bytes, measure: the copy cost can exceed the allocation it was meant to avoid.

ref struct is a scalpel, not a default. Its lifetime constraints are contagious — once a method takes or returns a span, callers inherit the stack-only discipline. Reach for it in parsing, serialization, and buffer-processing hot paths; do not thread it through general business logic where it merely fights you.

readonly struct should be the default for value types. It costs nothing, documents immutability, and eliminates the defensive-copy tax for free. Plain mutable `struct` is the exception, justified only by measured need (e.g. a struct enumerator that must advance).

Box-hunting is a JIT-disassembly skill. When a value-type-heavy loop allocates unexpectedly, suspect an interface conversion, a `params object[]`, an `IEnumerable<T>` enumeration, or a `Nullable<T>` flowing through `object`. A disassembler showing `box` at the call site is the ground truth; allocation profilers point you at the line, the IL tells you why.

The through-line of this chapter — and of Part I — is that “value vs reference” is not about memory location but about *copy semantics, identity, lifetime, and dispatch*. C# 12 through 14 did not change that model; they gave you sharper tools to express it: primary constructors and full-type `using` aliases (C# 12), `allows ref struct` and `Lock` (C# 13), and `field`, extension members, unbound `nameof`, and first-class spans (C# 14). The next chapter builds on this foundation to dissect records and `with`-expressions, where value semantics meet reference types head-on.

Sources: [What’s new in C# 14 — Microsoft Learn](#), [Implicit Span conversions in C# 14 — Start Debugging](#), [C# 14 The field Keyword — Thomas Claudius Huber](#).

2 · Records, Immutability, and Value Semantics

Records are the C# answer to a recurring need: types whose *identity is their data*. This chapter dissects the three record flavors, the exact code the compiler synthesizes on your behalf, the freeze-point semantics of `init` and `required`, and the inheritance traps that silently break value equality. We finish with the harder design question — when a record is the wrong tool and a plain `readonly struct` or a hand-rolled immutable aggregate serves you better.

The three flavors and what they actually are

C# 9 (.NET 5) introduced `record` (a reference type). C# 10 (.NET 6) added `record struct` and `readonly record struct` (value types). The keyword does not create a new kind of type — a `record class` is a `class`, a `record struct` is a `struct`. What the keyword *does* is instruct the compiler to synthesize a package of members that together give the type **value-based equality** and **non-destructive mutation**.

```
public record class Customer(string Name, string
    Email);           // reference type, value
                    equality
public record struct Point(int X, int Y);
    // value type, mutable fields
public readonly record struct Vec3(float X, float Y,
    float Z); // value type, immutable
```

The distinction that matters in practice:

- `record class` is heap-allocated, copied by reference, but compared by value. Two distinct `Customer` instances with the same `Name` / `Email` are `Equals`. This is the workhorse for DTOs and domain values that you pass around freely.
- `record struct` is a value type whose positional members compile to **mutable** auto-properties (`get` / `set`). It gives

you value equality *for free* — but it is not immutable. `p.X = 5;` compiles.

- `readonly record struct` is the one you almost always want when you reach for a value-type record: the positional members become `get / init`, the whole struct is `readonly`, and `with` is your only mutation path.

A subtle and frequently-missed point: a non-`readonly record struct` synthesizes its equality off mutable state. That is legal but dangerous if the struct is ever used as a dictionary key — mutate a field after insertion and the hash bucket no longer matches. Default to `readonly record struct` and only drop `readonly` when you have a measured reason.

What the compiler synthesizes

For a `record class C(...)`, the compiler generates (unless you write a member yourself, in which case it backs off):

1. A **primary constructor** matching the positional parameters, assigning each to a public `get / init` property of the same name.
2. `bool Equals(C? other)` plus an override of `object.Equals`, plus `== / !=` operators, all comparing every instance field via `EqualityComparer<T>.Default`.
3. `override int GetHashCode()` combining the same fields.
4. `override string ToString()` delegating to a synthesized `protected virtual bool PrintMembers(StringBuilder)`.
5. `void Deconstruct(out ...)` for positional records.
6. A **protected “copy constructor”** `C(C original)` and a `virtual C <Clone>()` method (the real name is unspeakable in C#) that powers `with`.
7. A `protected virtual Type EqualityContract { get; }` returning `typeof(C)` — the linchpin of inheritance-safe equality (see below).

Every one of these is *opt-out*. Write your own `Equals`, `GetHashCode`, `ToString`, or `Deconstruct` and the compiler

yields. This is what makes records composable rather than magic: you can customize one facet without losing the rest.

For value-type records the shape is almost identical, with two differences forced by the type system: there is no copy constructor (`with` copies the whole struct by value), and

`EqualityContract` is absent — a `struct` is implicitly sealed, so runtime-type comparison is unnecessary.

A validated record class with `init` and `required`

Positional parameters are convenient but they bypass validation — a positional `string Email` will accept `""`. The idiomatic fix is to keep the record positional for the simple fields and add explicitly-declared members for the ones that need invariants.

Here `Email` is a small validated value object, and `Tier` is a `required` member that the caller *must* set.

```

using System.Diagnostics.CodeAnalysis;

public readonly record struct EmailAddress
{
    public string Value { get; }

    public EmailAddress(string value)
    {
        if (string.IsNullOrEmpty(value) ||
            !value.Contains('@'))
            throw new ArgumentException($"Invalid
            email: '{value}'"", nameof(value));
        Value = value;
    }

    public override string ToString() => Value;
}

public enum Tier { Free, Pro, Enterprise }

public record class Customer(string Name, EmailAddress
    Email)
{
    // An explicitly-declared positional property with
    validation in its accessor.
    // C# 14's `field` keyword (introduced C# 14 /
    .NET 10) names the compiler-
    // synthesized backing field, so we can validate
    inside the `init` accessor
    // body without declaring a separate backing field
    ourselves. The primary
    // constructor routes its `Name` argument through
    this `init` setter, so
    // `value` here is the positional argument.
    public string Name
    {
        get;
        init => field =
            !string.IsNullOrEmpty(value)
                ? value.Trim()
                : throw new ArgumentException("Name
                required", nameof(value));
    }

    // A required member: the compiler forces every
    construction path to set it.
    public required Tier Tier { get; init; }

    // Non-positional derived/computed state stays
    immutable and lazy-free.

```



```

    public bool IsPaid => Tier is not Tier.Free;
}

```

Note what is happening with `Name`. A positional record already declares a `Name` property; here we *redeclare* it with a custom `init` accessor. The compiler detects the explicit declaration, uses our version, and still routes the primary-constructor parameter `Name` through it: positional construction assigns each parameter to its corresponding property, so our `init` setter runs with `value` bound to the constructor argument. Inside the accessor, the C# 14 `field` keyword names the synthesized backing field, letting us store the trimmed, validated value without hand-writing a backing field. (Before C# 14 you would declare an explicit `private readonly string _name;` field and assign it here.)

Construction now looks like this:

```

var c = new Customer(" Ada Lovelace ", new
    EmailAddress("ada@analytical.engine"))
{
    Tier = Tier.Pro
};
// c.Name == "Ada Lovelace" (trimmed)
// c.IsPaid == true

// Compiler error CS9035: required member
// 'Customer.Tier' must be set:
// var bad = new Customer("Ada", new
//     EmailAddress("a@b.c")); // ❌ won't compile

```

`required` (C# 11 / .NET 7) is a *compile-time* contract enforced at every call site that constructs the type without an attributed constructor. It composes with `init`: the member can only be set during object-initializer construction, and it *must* be. That combination — `required` + `init` — is the closest C# gets to “mandatory and then frozen.”

The freeze point: `init` and object initializers

`init` accessors (C# 9) are settable only during the object-initialization phase: inside the constructor, inside the object initial-

izer `{ }`, and inside a `with` expression. After the initializer's closing brace, the object is frozen. There is no “half-built” window visible to other threads because the initializer runs before the `new` expression returns its reference.

This is why `required + init` is safe in concurrent code: the publishing reference is only handed out after the freeze point. Contrast with a settable property, where a partially-initialized object can leak.

C# 14 adds **null-conditional assignment**, which interacts cleanly with mutable record members and `with`-built graphs:

```
order?.Metadata?["traceId"] = correlationId; // C#  
14: assignment is skipped if order or Metadata  
is null  
buffer?[index] = value; //  
and indexer assignment too
```

The whole assignment — including evaluation of the right-hand side — is short-circuited when any link in the chain is null.

Before C# 14 you wrote an explicit `if (order is { Metadata: { } m }) m["traceId"] = ...;`

`with`-expressions and non-destructive mutation

`with` produces a *copy* with selected members changed. For a `record class` it invokes the synthesized copy constructor (shallow-copying every field), then applies the initializer to the new instance. For a value-type record it copies the struct by value and overwrites the named fields.

```

public readonly record struct Vec3(float X, float Y,
    float Z)
{
    public float LengthSquared => X * X + Y * Y + Z *
        Z;

    public static Vec3 operator *(Vec3 v, float s) =>
        v with { X = v.X * s, Y = v.Y * s, Z = v.Z * s
        };
}

// Rotate a vector about the Z axis by `radians`,
producing a new value.
static Vec3 RotateZ(Vec3 v, float radians)
{
    var (sin, cos) = (MathF.Sin(radians),
        MathF.Cos(radians));
    return v with
    {
        X = v.X * cos - v.Y * sin,
        Y = v.X * sin + v.Y * cos,
        // Z is unchanged - `with` carries it over
        from the source copy.
    };
}

var p = new Vec3(1f, 0f, 0f);
var q = RotateZ(p, MathF.PI / 2); // ≈ (0, 1, 0)
// p is untouched: non-destructive mutation.

```

The key semantic: `with` copies *first*, then assigns. Members you do not mention are carried over verbatim from the source. For `readonly record struct` the entire operation is on the stack — no allocation, no GC pressure — which is exactly why value-type records shine in tight numeric loops.

A `record class with` does allocate (a fresh heap object plus a shallow field copy). That is usually fine, but in a hot loop that rebuilds a large record millions of times you are generating garbage proportional to iterations. Profile before assuming `with` is free for reference records.

Customizing `ToString` while keeping value equality

`ToString` on a record routes through `PrintMembers`. Override `PrintMembers` (not `ToString`) when you want to change formatting but keep the `TypeName { ... }` envelope and, crucially, keep the synthesized equality intact. Overriding `ToString` directly works too but throws away the structured envelope.

```
public record class ApiKey(string Owner, string
    Secret)
{
    // Keep value equality and the synthesized
    // ToString shell,
    // but never print the secret. PrintMembers
    // returns true if it wrote anything.
    protected virtual bool
        PrintMembers(System.Text.StringBuilder
            builder)
    {
        builder.Append("Owner = ").Append(Owner);
        builder.Append(", Secret = ").Append(Secret is
            { Length: > 0 } ? "***" : "(empty)");
        return true;
    }
}

Console.WriteLine(new ApiKey("svc-billing",
    "sk_live_9f3a..."));
// ApiKey { Owner = svc-billing, Secret = *** }

// Equality is unaffected – it still compares the
// *real* Secret field:
var a = new ApiKey("svc", "sk_1");
var b = new ApiKey("svc", "sk_1");
Console.WriteLine(a == b); // True
```

In a **sealed** record or a record struct, declare `PrintMembers` as `private bool PrintMembers(...)`; in a non-sealed `record class` it must be `protected virtual` so derived records can chain into it. Get the signature wrong and the compiler quietly does not recognize it as the override hook — your method exists but `ToString` ignores it.

Inheritance, `EqualityContract`, and the sealed-equality trap

Records can inherit from records (never from a plain class, and a class can never inherit a record). Value equality across a hierarchy is where most production bugs hide.

The synthesized `Equals` does **not** just compare fields. It first compares `EqualityContract` — a `protected virtual Type` returning `typeof(ThisRecord)`. Because it is `virtual`, a `Base` instance and a `Derived` instance return *different* contracts, so `base.Equals(derived)` is `false` even if every shared field matches. This is correct: a `Manager` is not equal to an `Employee` with the same name.

The trap appears when you *break* this invariant. Consider a hierarchy where a derived record adds state but you compare through the base type:

```
public record class Employee(string Name)
{
    public decimal Salary { get; init; }
}

public record class Manager(string Name) :
    Employee(Name)
{
    public int ReportCount { get; init; }
}

Employee e = new Employee("Sam") { Salary = 100m };
Employee m = new Manager("Sam") { Salary = 100m,
    ReportCount = 4 };

Console.WriteLine(e == m); // False -
    EqualityContract differs (Employee vs
    Manager). Correct.
Console.WriteLine(m == m with { ReportCount = 9 });
    // False - Manager-aware Equals sees
    ReportCount. Correct.
```

So far so good — the *virtual* `EqualityContract` saves you. Note one detail: overriding `EqualityContract` *alone* on the base does **not** break this, because each derived record re-synthesizes

its own `protected override Type EqualityContract => typeof(Derived)`. The genuine bug is the **fixed-equality trap**: developers sometimes try to “fix” cross-hierarchy comparison by hand-rolling `Equals` / `GetHashCode` on the base so they compare only the shared fields and ignore the runtime type. Because you have taken over `Equals` entirely, the compiler stops synthesizing the type-aware comparison, and every derived record inherits the broken base comparison through `base.Equals`. The result: a `Manager` can equal an `Employee`, silently corrupting hash sets and dictionaries keyed on the base type:

```
// ❌ ANTI-PATTERN – do not do this.
public record class BadBase(string Name)
{
    // Hand-rolling Equals/GetHashCode to ignore the
    // runtime type defeats the
    // contract check. Derived records inherit this
    // via base.Equals, so a
    // BadBase and any subclass with the same Name now
    // compare equal – a
    // HashSet<BadBase> will collide subtypes that
    // should be distinct.
    public virtual bool Equals(BadBase? other) =>
        other is not null && Name == other.Name;
    public override int GetHashCode() =>
        Name.GetHashCode();
}
```

The correct posture is: **let the compiler own `EqualityContract`**. If you genuinely want all subtypes to be interchangeable for equality, that is a design smell — you probably want a single non-inherited record plus a discriminator field, or a closed hierarchy you compare explicitly. The one *safe* customization is `sealed`-ing the leaf records: a `sealed record` has an effectively-sealed `EqualityContract`, which is fine because no further derivation can occur.

When you *do* need a polymorphic value-equal hierarchy (the classic discriminated-union shape), seal every leaf and never touch `EqualityContract`:

```

public abstract record class Shape;
public sealed record class Circle(double Radius)
    : Shape;
public sealed record class Rectangle(double W, double
    H)    : Shape;

Shape s1 = new Circle(2.0);
Shape s2 = new Circle(2.0);
Console.WriteLine(s1 == s2); // True
Console.WriteLine(s1 == new Rectangle(2, 2)); // False
           - contracts differ

```

This is the only inheritance shape with records I recommend in domain code: an `abstract` base with `sealed` leaves, equality untouched.

Required members with positional records and

`[SetsRequiredMembers]`

You **cannot** put `required` on a positional parameter directly — but you can redeclare the positional property with `required`. When you do, the synthesized primary constructor is automatically decorated with `[SetsRequiredMembers]` (the compiler knows the primary ctor sets them), so positional construction stays legal while object-initializer construction still demands the required member.

The two consumption paths, side by side:

```

using System.Diagnostics.CodeAnalysis;

public record class Invoice
{
    public required string Number { get; init; }
    public required decimal Amount { get; init; }
    public string Currency { get; init; } = "USD";

    // A convenience constructor that initializes ALL
    // required members.
    // [SetsRequiredMembers] (C# 11) tells the
    // compiler "trust me, these are set",
    // so callers of this ctor are NOT forced to use
    // an object initializer.
    [SetsRequiredMembers]
    public Invoice(string number, decimal amount,
        string currency = "USD")
    {
        Number = number;
        Amount = amount;
        Currency = currency;
    }

    // Records also synthesize a copy constructor for
    // `with`; it must propagate
    // the attribute, which the compiler handles
    // automatically for records.
}

// Path 1 – object initializer: required members
// enforced at compile time.
var i1 = new Invoice { Number = "INV-001", Amount =
    42.50m }; // ✓
// var bad = new Invoice { Number = "INV-002" };
// ✗ CS9035: Amount not set

// Path 2 – attributed constructor: no initializer
// required.
var i2 = new Invoice("INV-003", 99.00m, "EUR");
// ✓

```

`[SetsRequiredMembers]` is **trust-based**: the compiler does not verify the constructor actually assigns every required member. If you decorate a constructor that forgets one, you have created a silent invariant hole. Audit these by hand or with an analyzer. The attribute is the right tool when a record needs both a convenient positional/imperative constructor *and* the initializer-based guarantee for other call sites.

A worked example: an immutable aggregate

Records give you equality and copying, but they do **not** make referenced collections immutable. The single most common immutability bug is putting a `List<T>` inside a record and believing the record is now safe to share. The following aggregate is genuinely immutable: it stores `ImmutableArray<T>`, validates in construction, and exposes only non-destructive transformations. Collection expressions (C# 12) keep the construction terse.

```

using System.Collections.Immutable;
using System.Linq;

public sealed record class ShoppingCart
{
    public required string CustomerId { get; init; }
    public ImmutableArray<LineItem> Items { get; init; }
        = []; // C# 12 collection expression

    public decimal Total => Items.Sum(li =>
        li.LineTotal);
    public int Count => Items.Length;

    // Non-destructive "mutation": returns a new cart,
    // never touches `this`.
    public ShoppingCart Add(LineItem item) =>
        this with { Items = Items.Add(item) };

    public ShoppingCart Remove(string sku) =>
        this with { Items = [.. Items.Where(li =>
            li.Sku != sku)] }; // spread into a new array

    // ImmutableArray<T>.Equals compares the *backing
    // array by reference*, not by
    // content – so two carts with identical items but
    // different array instances
    // would be unequal under the synthesized record
    // equality. Override Equals and
    // GetHashCode to restore true value semantics
    // over the item contents.
    public bool Equals(ShoppingCart? other) =>
        other is not null
        && CustomerId == other.CustomerId
        && Items.SequenceEqual(other.Items);

    public override int GetHashCode()
    {
        var hash = new HashCode();
        hash.Add(CustomerId);
        foreach (var item in Items)
            hash.Add(item);
        return hash.ToHashCode();
    }
}

public readonly record struct LineItem(string Sku,
    decimal UnitPrice, int Qty)
{
    public decimal LineTotal => UnitPrice * Qty;
}

```

```
// --- usage ---
var empty = new ShoppingCart { CustomerId = "c-1" };
var withApple = empty.Add(new LineItem("APPLE",
    0.50m, 6));
var withBanana = withApple.Add(new LineItem("BANANA",
    0.30m, 12));

Console.WriteLine(empty.Count);           // 0 - original
                                           never mutated
Console.WriteLine(withBanana.Total);      // 6.60
Console.WriteLine(withApple ==
    withBanana.Remove("BANANA"));        // True - with
                                           the custom content-based Equals
```

Two details earn their keep. First — and this is the trap — `ImmutableArray<LineItem>` does **not** participate in record equality the way intuition suggests. Its `IEquatable<ImmutableArray<T>>.Equals` (and `==`) compares the *backing array by reference*, i.e. whether both values wrap the same `T[]` instance; it is **not** element-by-element. The compiler-synthesized `ShoppingCart.Equals` calls `EqualityComparer<ImmutableArray<LineItem>>.Default.Equals`, which dispatches to that reference-based comparison. So without intervention, `withApple == withBanana.Remove("BANANA")` would be **False** even though the contents match — `Remove` builds a fresh backing array. The hand-written `Equals` / `GetHashCode` above restores genuine value semantics by comparing items with `SequenceEqual` and folding them into a `HashCode`. (Whenever you override `Equals` you must override `GetHashCode` to keep the equal-objects-share-a-hash invariant — otherwise hash-based collections break.) Second, every transformation returns a fresh `ShoppingCart`; the `with` expression reuses every unchanged field, so the cost is one small heap object plus an `ImmutableArray` structural-sharing operation, not a deep clone.

If you had used `List<LineItem>` here, `Add` could mutate the shared list under another holder of the same cart, and equality would still compare list *references*, not contents — the same reference-equality pitfall, but now compounded by shared mutable state. `ImmutableArray<T>` (or `ImmutableList<T>` for frequent middle-insertions) is the correct *immutable backing store*; the

custom `Equals` is what supplies the *content equality* records do not give you for free over collection fields.

Extension members on records (C# 14)

C# 14 (.NET 10) introduced **extension blocks** —

`extension(Type instance) { ... }` — which add extension *properties* and static extension members, not just methods.

They are a clean way to layer query helpers onto records you do not own without polluting the record's own surface:

```
public static class CartExtensions
{
    extension(ShoppingCart cart)                // C#
        14 extension block
    {
        // An extension *property* — impossible before
        C# 14.
        public bool IsEmpty => cart.Count == 0;
        public LineItem? Largest =>
            cart.Items.IsEmpty ? null :
            cart.Items.MaxBy(i => i.LineTotal);
    }
}

// usage
if (!withBanana.IsEmpty)
    Console.WriteLine(withBanana.Largest);
```

This keeps the record itself focused on state and invariants while presentation/query concerns live in extensions — a good separation for DTOs that travel across layers.

Choosing the right tool: records, structs, or neither

Use case	Reach for	Why
Wire/API DTO, message, event	<code>record class</code>	Value equality for caching/dedup; reference semantics for cheap passing; nullable + <code>required</code> express contracts
Small immutable domain value (money, coordinate, color)	<code>readonly record struct</code>	No allocation, stack-copied <code>with</code> , value equality; $\leq 16\text{--}24$ bytes
Mutable scratch value type in a hot loop	plain <code>struct</code> (no record)	You do not want synthesized equality off mutable state
Large value type (many fields, big embedded data)	<code>record class</code> or plain <code>class</code>	Avoid per-copy cost of value types and per- <code>with</code> struct copies
Closed discriminated union	<code>abstract record + sealed</code> leaves	Polymorphic value equality done correctly

The decisive question for value-type records is **size and copy frequency**. A `with` on a value-type record copies the *entire* struct. For a `Vec3` (12 bytes) that is a register or two — essentially free. For a `record struct` with twenty fields, several `decimal`s (16 bytes each) and a couple of inline `Guid` fields (16 bytes each), every `with` copies the whole thing, and passing it to methods copies it again. (Note: a *reference-typed* field such as a `Guid[]` only adds a pointer to the struct's size — what blows up the copy cost is embedded *value* data, not references.) Past roughly 24 bytes the value-type advantage inverts: you pay copy cost everywhere, and the JIT can no longer keep it in registers. Measure with `BenchmarkDotNet`; the crossover depends on ac-

cess patterns, but “large value type + frequent `with`” is a reliable performance pitfall.

For reference-type records the cost model is the opposite: cheap to pass (one pointer), but every `with` allocates. In a request/response path that is negligible; in a per-frame game loop or a million-row transform it is not. When `with`-churn on a `record class` shows up in GC traces, the fix is usually to batch the mutations (build once, freeze once) rather than chaining many `with` expressions.

Pitfalls, condensed

- **Mutable collection inside a record.** A `record` with a `List<T>` field is *not* immutable and does *not* have content equality on that field — it compares the list by reference. Back immutable aggregates with `ImmutableArray<T>` / `ImmutableList<T>` and treat the record as a value.
- **Equality across an inheritance hierarchy without understanding `EqualityContract`.** The virtual contract is what makes `Employee` $\neq$ `Manager`. Never hand-roll `Equals` / `GetHashCode` on a base record to ignore the run-time type — derived records inherit the broken comparison through `base.Equals` and collide in hash sets. If you need subtypes interchangeable, your model is wrong. Prefer `abstract` base + `sealed` leaves.
- **`record struct` for large value types.** Every `with` and every method call copies the whole struct. Beyond ~24 bytes, switch to a `record class` or restructure; do not assume value types are always cheaper.
- **`[SetsRequiredMembers]` on a constructor that forgets a member.** Trust-based, unverified by the compiler — a silent invariant hole. Audit attributed constructors.
- **Overriding `ToString` instead of `PrintMembers`.** Works, but discards the structured `TypeName { ... }` envelope and the derived-record chaining. Override `PrintMembers`

(right access modifier for sealed vs. open) to customize formatting while keeping equality and the envelope intact.

- **Non-readonly record struct as a dictionary key.**
Mutating a field after insertion desynchronizes the hash bucket. Default to `readonly record struct`.

Records remove an enormous amount of boilerplate, but the synthesized members encode *opinions* — about equality, about copying, about the runtime type. The expert move is knowing exactly what was generated, where the compiler backs off to your code, and where value semantics quietly stop matching your mental model of immutability.

Sources: - [The `field` contextual keyword — Microsoft Learn](#) - [Records — C# reference, Microsoft Learn](#) - [required modifier — Microsoft Learn](#) - [Resolve errors and warnings related to record declarations — Microsoft Learn](#)

3 · Pattern Matching, List Patterns, and Switch Expressions

Pattern matching is the load-bearing wall of modern C# control flow: a single declarative grammar that subsumes type tests, deconstruction, range checks, and structural matching, and that the compiler lowers into branch-table-quality IL. This chapter treats patterns as a first-class type-system feature — how each pattern composes, what the compiler emits, where it allocates or throws, and how to use exhaustiveness as a static safety net rather than a runtime surprise.

The pattern grammar, end to end

A *pattern* is a syntactic form that tests an input expression and optionally binds parts of it. C# has accumulated patterns across many releases, and they all compose: a property pattern can contain a relational pattern that contains a constant, a positional pattern can contain a list pattern, and so on. The full set you will use in C# 14 / .NET 10:

Pattern	Example	Introduced
Constant	<code>42, "GET", null</code>	C# 7.0
Declaration	<code>string s</code>	C# 7.0
Type	<code>string</code> (no binding)	C# 9.0
<code>var</code>	<code>var x</code>	C# 7.0
Discard	<code>_</code>	C# 8.0
Property	<code>{ Length: 0 }</code>	C# 8.0
Positional	<code>(0, var y)</code>	C# 8.0
Relational	<code>> 0, <= 100</code>	C# 9.0
Logical (<code>and</code> / <code>or</code> / <code>not</code>)	<code>>= 200 and < 300</code>	C# 9.0
Parenthesized	<code>(> 0) or (< -10)</code>	C# 9.0
Extended property	<code>{ Address.City: "Praha" }</code>	C# 10.0
List	<code>[1, 2, 3]</code>	C# 11.0
Slice	<code>[first, .., last]</code>	C# 11.0

Nothing in C# 12, 13, or 14 *adds a new pattern kind*, but the surrounding language changed how patterns are reached: C# 14's first-class implicit `Span<T> / ReadOnlySpan<T>` conversions and the `field` keyword both interact with patterns in ways worth knowing. (A common source of terminology confusion: C# 13's new `System.Threading.Lock` type makes the compiler recognize it specially in a `lock` statement — that “lock object” recognition is not part of the *matching* grammar at all; don't conflate the two.) The matching grammar itself stabilized at C# 11 with list patterns, and slice patterns arrived alongside them in the same release.

The two host contexts are the `is` expression (a boolean test that may bind) and the `switch` expression / statement (multi-arm dispatch). Both share the identical pattern grammar.

Declaration, type, and constant patterns

The declaration pattern tests runtime type and binds on success. The type pattern (C# 9) does the same test without a binding — useful when you only need the branch, not the value.

```
static string Describe(object o) => o switch
{
    null                => "null reference",    //
        constant pattern
    string { Length: 0 } => "empty string",    // type
        + property
    string s            => $"string of {s.Length}", //
        declaration pattern, binds s
    int                 => "some int",          //
        type pattern, no binding
    -                   => "other"
};
```

A declaration pattern `T x` succeeds when the input is non-null and reference-convertible (or boxing/unboxing-convertible) to `T`; it never performs a user-defined conversion. The `is` operator's long-standing null behavior is preserved: `null is object` is `false`, and `null is string s` does not bind. This is the single most common reason a `null` arm must come *first* in a switch — without it, a `null` input falls through to the discard or throws.

Constant patterns compare with `==` semantics for the operand type, with two deliberate exceptions. First, `null` is matched without ever invoking a user-overloaded `==` (the compiler guarantees `x is null` is a reference/ `HasValue` test). Strings compare ordinarily. Second — and this is the trap experts get backwards — the constant pattern `double.NaN` (or `float.NaN`) is **special-cased to match `NaN`**, even though `NaN == NaN` is `false` under IEEE 754. So `x is double.NaN` returns `true` when `x` is `NaN`, behaving like `double.IsNaN(x)`, *not* like `x == double.NaN`. This is the one constant pattern that intentionally diverges from `==`:

```
static string Classify(double x) => x switch
{
    < -4.0      => "too low",
    > 10.0     => "too high",
    double.NaN => "unknown", // MATCHES a NaN input
                        - equivalent to double.IsNaN(x)
    -          => "acceptable"
};
// Classify(double.NaN) => "unknown"
```

Note the asymmetry with *relational* patterns: `< double.NaN` or `> float.NaN` is a compile-time error (a NaN bound can never order-compare), but the NaN *constant* pattern is legal and meaningful. Outside a pattern, `x == double.NaN` is still always `false` — use the pattern or `double.IsNaN(x)`, never the bare equality.

Relational and logical patterns: classifying an HTTP result

Relational patterns (`<`, `<=`, `>`, `>=`, C# 9) compare against a constant. Logical patterns (`and`, `or`, `not`, C# 9) combine patterns — and crucially, `and` / `or` operate on *patterns*, not boolean expressions, so the input is mentioned once and the compiler can collapse the comparisons into a range test.

```
using System.Net;

static string Classify(HttpStatusCode code) =>
    (int)code switch
{
    < 100                => "invalid",
    >= 100 and < 200     => "informational",
    >= 200 and < 300     => "success",
    >= 300 and < 400     => "redirect",
    301 or 308          => "permanent redirect", //
                        unreachable - see below
    >= 400 and < 500     => "client error",
    429                 => "rate limited", //
                        also unreachable
    >= 500 and < 600     => "server error",
    -                    => "unknown"
};
```

This compiles, but the compiler emits **CS8510** (“**The pattern is unreachable...**”) for the dead arms `301 or 308` and `429`: each is already subsumed by an earlier range arm (`>= 300 and < 400`, `>= 400 and < 500`). (CS8510 is the *switch-expression* form; the *switch-statement* equivalent for a subsumed `case` label is CS8120 — don’t confuse the two.) This is the *arm-shadowing pitfall* in its purest form — a broad pattern placed before a specific one renders the specific one dead. The fix is to order specifics before generals:

```
static string Classify(int code) => code switch
{
    429                => "rate limited",
    301 or 308         => "permanent redirect",
    >= 200 and < 300   => "success",
    >= 300 and < 400   => "redirect",
    >= 400 and < 500   => "client error",
    >= 500 and < 600   => "server error",
    >= 100 and < 200   => "informational",
    -                  => "unknown"
};
```

For an integer switch dense in a small range like this, the compiler lowers the constant arms (`429`, `301`, `308`) into a **jump table** (`switch` IL opcode) and the range arms into a short chain of compares. Note `not : code is not (>= 200 and < 300)` reads as “outside the 2xx range” and is exactly `code < 200 || code >= 300` after lowering — but it states intent structurally and cannot drift if the bounds change.

A real gotcha with `and`: precedence. `not` binds tightest, then `and`, then `or`. So `1 or 2 and not 3` parses as `1 or (2 and (not 3))`. When mixing, parenthesize: `(1 or 2) and not 3`. Relational patterns require a compile-time constant on the right — `> length` where `length` is an ordinary (non-`const`) local does **not** compile; only literals, `const` fields/locals, and `enum` members qualify. Relational patterns are not arbitrary comparisons; to compare against a runtime value, use a `when` guard.

Property patterns, nested destructuring, and the extended form

Property patterns (C# 8) match by recursively applying sub-patterns to members. The extended property pattern (C# 10) lets you reach through a member chain with dotted syntax instead of nesting braces.

```
record Address(string City, string Country);
record Person(string Name, int Age, Address Address);

static bool IsPragueAdult(Person p) => p is
{
    Address.City: "Praha",    // extended (C# 10):
                             // equivalent to Address: { City: "Praha" }
    Age: > 18                 // relational sub-pattern
};
```

The two forms are exactly equivalent after parsing — `{ Address.City: "Praha" }` lowers to `{ Address: { City: "Praha" } }`. Prefer the dotted form for single-deep reaches and the nested-brace form when you bind multiple members of the same sub-object, because the nested form evaluates `Address` once:

```
static string Greet(Person p) => p switch
{
    { Address: { City: "Praha", Country: "CZ" }, Age:
      var a } => $"Pražák, {a}",
    { Address.City: var c } => $"From {c}",
    _ => "unknown"
};
```

Property patterns are null-safe at every hop: `{ Address.City: "Praha" }` returns `false` (does not throw) if `p` is null or `p.Address` is null. The compiler inserts the null checks for you; this is one of the highest-value reasons to prefer a property pattern over hand-written `p?.Address?.City == "Praha"` — the latter has the same null safety but cannot bind, and a pattern can.

A property pattern accessing a member calls the property getter. The decision tree Roslyn lowers a switch into evaluates each distinct access path *once per node* and threads the temp through the arms that branch from it, so a getter probed by several arms along the *same* path is typically called once, not once per arm. But the getter is still re-invoked for inputs that take a *different* branch and reconverge, and it is always called at least once when the path is reached — so a getter that lazily materializes, mutates, or does I/O is a hazard inside a pattern. Keep pattern-probed members cheap, idempotent, and side-effect-free; the compiler does no purity analysis and will happily call an expensive getter.

In C# 14 the `field` keyword (field-backed properties without an explicit backing field) does not change pattern semantics — patterns still go through the getter — but it makes it trivial to add validation in a setter while keeping a pattern-friendly getter, e.g. `public int Age { get => field; set => field = value < 0 ? 0 : value; }.`

Positional patterns over Deconstruct and tuples

A positional pattern `(p1, p2, ...)` matches by deconstruction. It works on any type with an accessible `Deconstruct` method (or extension), on `System.Tuple / ValueTuple`, and on records (which generate `Deconstruct` from their primary constructor parameters).

```

readonly record struct Point(int X, int Y);

static string Quadrant(Point p) => p switch
{
    (0, 0)                => "origin",
    (> 0, > 0)            => "I",
    (< 0, > 0)            => "II",
    (< 0, < 0)            => "III",
    (> 0, < 0)            => "IV",
    (_, 0) or (0, _)      => "on axis",
    _                     => "unreachable" // logically
                                     // dead, but the compiler can't prove
                                     // relational-pattern coverage, so CS8509 without
                                     // it
};

```

Positional and property sub-patterns combine freely. You can mix positions and named members and bind in the same pattern:

```

static string Plot((int X, int Y) p) => p switch
{
    (var x, 0) => $"x-axis at {x}", // bind X,
                                     match Y == 0
    { Y: var y } when y == p.X => $"diagonal {y}",
    _ => "elsewhere"
};

```

Note the `when` clause: it is an arbitrary boolean guard evaluated after the pattern matches and bindings are in scope. Guards are the escape hatch for conditions the pattern grammar cannot express (relations between two bound values, method calls), but they **defeat exhaustiveness analysis and jump-table lowering** — an arm with a `when` clause is always treated as possibly-not-matching, so the compiler cannot prove a switch exhaustive on the strength of a guarded arm, and the arm lowers to a linear test. Use guards sparingly on hot paths.

If a type exposes multiple `Deconstruct` overloads, the positional pattern's *arity* (number of sub-patterns) selects the matching overload; two overloads of the same arity are an ambiguity error at the pattern site. ([\[OverloadResolutionPriority\]](#) , intro-

duced in C# 13, tunes ordinary method overload resolution — it does not disambiguate same-arity `Deconstruct` overloads for a pattern.)

List patterns and slice patterns

List patterns (C# 11) match countable, indexable sequences by shape. The slice pattern `..` matches zero or more elements and may itself bind a sub-range. The canonical command parser:

```
static string Interpret(string[] argv) => argv switch
{
    [] => "no command",
    ["help"] => "show help",
    ["move", var dir, .. var rest]
        => $"move {dir}" + (rest.Length > 0 ? $" then
            {string.Join('/', rest)}" : ""),
    ["echo", .. var words] => string.Join(' ',
        words),
    [var single] => $"unknown unary
        command '{single}'",
    - => "unrecognized"
};

// Interpret(["move", "north", "fast", "quietly"])
// => "move north then fast/quietly"
```

`[first, .., last]` (a *slice pattern*) is the idiom for “match endpoints, ignore the middle.” When the slice binds (`.. var rest`), the compiler emits a call to the type’s slice accessor — for `T[]` that is `RuntimeHelpers.GetSubArray` (which **allocates** a new array), and for `string` it is `Substring`. For `Span<T>` / `ReadOnlySpan<T>` it is the `Slice` method, which is **O(1) and allocation-free** because a span slice is just a re-based pointer + length. This asymmetry is the central performance fact about slice patterns.


```

static bool PalindromicEndpoints(ReadOnlySpan<int>
    data) => data switch
{
    [] or [_] => true, // 0 or
    // 1 element: trivially symmetric ends
    [0, .., 0] => true, // first
    // and last are both 0
    [var a, .., var b] => a == b, //
    // general endpoint equality
};

// PalindromicEndpoints([0, 7, 3, 0]) => true
// (matches [0, .., 0])
// PalindromicEndpoints([5, 1, 5]) => true
// (matches [var a, .., var b], 5==5)
// PalindromicEndpoints([1, 2, 3]) => false

```

This switch is exhaustive without a `_` arm: `[]`, `[_]`, and `[var a, .., var b]` together cover lengths 0, 1, and ≥ 2 , and the compiler proves it. The unused `..` in `[var a, .., var b]` does not bind, so **no slice is taken and nothing allocates** — the only operations emitted are a length read and two indexer reads (`data[0]` and `data[^1]`). The C# 14 first-class implicit span conversions mean you can pass an `int[]` or a collection-expression `[0, 7, 3, 0]` directly to this method without `.AsSpan()`, because arrays now implicitly convert to `ReadOnlySpan<int>`.

Length/Count semantics and the lazy-sequence trap

A list pattern first reads the sequence's length, then probes indices. For `T[]` it reads `Length`; for types with `Count` / `Length` plus an indexer and a `Slice` / `this[Range]` it uses those (the type must be *countable* and *indexable*). The compiler picks the most specific member.

The trap: **list patterns are not defined on `IEnumerable<T>`**.

You cannot write `someEnumerable is [_, ..]` — there is no `Count` or indexer to satisfy the requirement. If you have an `IEnumerable<T>` and reach for a list pattern, you must first materialize it (`ToList()` / `ToArray()`), which enumerates the whole sequence — **eager, allocating, and fatal for infinite or expensive generators**:

```

IEnumerable<int> Naturals() { int i = 0; while (true)
    yield return i++; }

// var first = Naturals() is [var head, ..]; // does
// NOT compile – no Count/indexer
// var first = Naturals().ToList() is [var head, ..];
// COMPILES but hangs forever

```

Even for a finite-but-large or side-effecting `IEnumerable`, materializing just to pattern-match defeats laziness silently. Prefer `IReadOnlyList<T>` / arrays / spans at the boundary, or pull the head with `TryGetNonEnumeratedCount` + an explicit enumerator when you genuinely have a lazy source. Note that not every collection is indexable: `LinkedList<T>`, for instance, has *no* `this[int]` indexer, so a list pattern won't even compile against it — you'd have to copy to an array first. And for a custom collection that *does* expose `Count` plus an $O(n)$ indexer (a skip-list or a lazily-paged view, say), a multi-index list pattern like `coll is [_, _, _, var fourth]` re-walks from the front on each indexer call, turning a single shape test into $O(n^2)$ in the worst case. Match shape only where indexing is genuinely cheap (`T[]` , `List<T>` , `Span<T>` , `string`).

var patterns, discards, and capturing

The `var` pattern (`var x`) always succeeds (it is a pure binding, no type test) and captures the input — or a sub-component — into a new variable typed by inference. The discard `_` matches and binds nothing. These are how you reach into a structure and pull values out:

```

static int Manhattan(Point p) => p switch
{
    (var x, var y) => Math.Abs(x) + Math.Abs(y)
};

```

A `var` pattern is *not* a null check. `obj is var v` is always `true` even when `obj` is null, binding `v = null`. This differs from `obj is object o` (which is `false` for null) and is a frequent source of “why did my null slip through” bugs. Use `var`

when you’ve already excluded null upstream or genuinely accept null; use a declaration/type pattern when null must be rejected.

Captures are flatly forbidden beneath an `or` or `not` combinator (error **CS9135**: “A variable may not be declared within a ‘not’ or ‘or’ pattern”). The rule is structural, not name-based: even `(var a, _) or (_, var b)` is illegal — a designator under `or` would be assigned on only one branch, and under `not` on the failing branch, so the compiler bans them outright rather than reason about definite assignment. `and`, by contrast, *does* allow captures, because both operands must match: `[> 0, ..]` and `var whole` binds `whole` to the entire array only when the first element is positive. The `T x` declaration form combined with `and (Person { Age: > 18 } and var adult)` is the idiomatic “test then bind the whole thing.”

Switch expressions: arms, ordering, exhaustiveness

A switch expression evaluates arms **top to bottom** and yields the first match. There is no fall-through, no `break`; the result is a value. Arm order is therefore semantically significant whenever patterns overlap — the compiler will warn (unreachable arm) only when an earlier arm *fully subsumes* a later one, but it will *not* warn when an earlier arm merely shadows part of a later arm’s domain. That silent partial-shadowing is the subtle version of the ordering pitfall.

Exhaustiveness: a switch expression must, in principle, handle every input. When the compiler cannot prove totality it emits **CS8509** (“does not handle all possible inputs”) and inserts a runtime guard that throws

`System.Runtime.CompilerServices.SwitchExpressionException` on an unmatched input. This is the most important behavioral difference from a switch *statement*, which simply does nothing on no-match. A non-exhaustive switch expression is a latent `throw`, not a compile error — unless you elevate it.

```

abstract record Shape;
sealed record Circle(double Radius) : Shape;
sealed record Square(double Side) : Shape;
sealed record Rectangle(double W, double H) : Shape;

static double Area(Shape s) => s switch
{
    Circle c    => Math.PI * c.Radius * c.Radius,
    Square sq   => sq.Side * sq.Side,
    // Rectangle intentionally omitted
};
// warning CS8509: The switch expression does not
//   handle all possible values
// of its input type. For example, the pattern
//   'Rectangle' is not covered.

```

Note the compiler **names the missing case** (`Rectangle`) — this is the single best reason to model closed domains as a sealed hierarchy and switch over it without a `_` arm: add a new derived type, and every non-exhaustive switch lights up at compile time with the exact gap. Adding `Rectangle r => r.W * r.H` removes *that named gap* from the diagnostic. Be precise about what does **not** happen, though: C# does not perform closed-hierarchy exhaustiveness analysis. It treats any class or record base as an *open* type for switch purposes — even an `abstract` base with all-`sealed` leaves and a private constructor — so a residual CS8509 (plus the runtime guard and a possible `null`) survives even after every subtype you wrote is covered. The named-gap diagnostic is still enormously useful as a regression signal; just don't expect the warning to reach zero or the `SwitchExpressionException` path to be dead-stripped purely from leaf coverage. See the caveats below.

Two caveats that bite experts:

1. **No type hierarchy is “closed” to the exhaustiveness checker.** C# treats every class and record base as an *open* type for switch purposes — even an `abstract` base with all-`sealed` leaves and a private constructor. (Records make this concrete: the compiler-generated `protected` copy constructor literally permits external derivation. But the open-type treatment applies to plain classes too.) Combined with

the ever-present possibility of a `null` input, the compiler emits CS8509 rather than treating the switch as exhaustive, and it keeps the throw guard, even when every subtype you wrote is matched. Adding an explicit `null =>` arm closes the null hole but not the external-derivation hole, so a residual CS8509 remains. The honest options are: (a) leave the residual CS8509 and promote it to an error (caveat 2) so it acts as a build gate while you keep the no-`_`-arm benefit, or (b) install a closed-hierarchy diagnostic suppressor (e.g. `ClosedTypeHierarchyDiagnosticSuppressor`) that recognizes the sealed-leaf invariant and silences CS8509 only when it can prove closure. Adding a `_` arm “fixes” the warning but is the worst choice — it permanently blinds you to new unhandled subtypes, which was the whole point of switching over the hierarchy.

2. Make non-exhaustiveness a build break for closed domains. Add

`<WarningsAsErrors>CS8509</WarningsAsErrors>` to the `.csproj`. Now an unhandled case fails the build instead of throwing in production. This turns the sealed-hierarchy switch into the C# equivalent of an ML compiler’s exhaustiveness checker.

```
<PropertyGroup>
    <WarningsAsErrors>$(WarningsAsErrors);CS8509;CS8524</WarningsAsErrors>
</PropertyGroup>
```

For switches over enums specifically, the related warning is CS8524 (input could be an out-of-range enum value not named by any member); handle it with a discard arm or by accepting that any `int`-backed enum can hold unnamed values.

Patterns in `is`, `while`, `for`, and other control flow

Patterns are not confined to `switch`. An `is` expression is usable anywhere a `bool` is — including loop conditions — and its

bindings leak into the enclosing scope on the true branch (definite-assignment analysis tracks this).

```
// Pattern in a while condition: drain a stack while
// the top is positive.
// The Stack<int>(IEnumerable<int>) ctor pushes in
// enumeration order, so the
// LAST element seeded ends up on top. Seeding [5, -1,
// 4, 1, 3] puts 3 on top.
var stack = new Stack<int>([5, -1, 4, 1, 3]); // top
// -> bottom: 3, 1, 4, -1, 5
while (stack.TryPeek(out var top) && top is > 0)
{
    Console.WriteLine($"{stack.Pop()} "); // prints: 3 1
    // 4 (stops at -1)
}

// Pattern in an if with binding that escapes to the
// rest of the method.
object payload = GetPayload();
if (payload is Person { Age: >= 18 } adult)
{
    Register(adult); // 'adult' is in scope and
    // definitely assigned here
}

// Pattern in a for loop guard.
for (object? cur = head; cur is Node { Next: var next,
    Value: var v }; cur = next)
{
    Process(v);
}
```

The binding-escape rule turns on *definite assignment*, not lexical position. A variable declared by `is` is in scope from its declaration to the end of the enclosing block, but you may only read it where the compiler can prove it was assigned — i.e. where the pattern *must have* matched. For `if (x is string s) { ... }`, `s` is assigned inside the body. After the `if`, `s` is in scope but **not** definitely assigned (you reach there both when the match succeeded *and* when it failed), so you can't read it. The mirror image is the guard clause `if (x is not string s) return;`: there `s` is *not* assigned inside the body (the match failed), but it **is** definitely assigned after the `if`, because the only way to fall through is for the positive match to have hap-

pened. This is the same flow analysis that powers `is not null` narrowing for nullable reference types: `if (x is not null) { /* x is non-null here */ }`.

One caveat the flow rules expose: a binding declared by `is` leaks its *name* into the enclosing scope even on paths where it is unassigned, so you cannot reuse that identifier for a different declaration later in the same block. The name is reserved by the declaration's scope, not by whether the match succeeded.

Implicit-index access, `Length / Count`, and `^` in patterns

List patterns are built on the index/range machinery. `[.., var last]` is sugar over `input[^1]` after a length check; `[var first, ..]` over `input[0]`. The compiler reads `Length / Count` exactly once per match and computes from-end indices relative to it. You can also use `^` (from-end index) *outside* patterns inside object initializers — C# 13 added implicit-index `^` access in object initializers, but only for a collection/array *member being initialized*, not for a bare array-creation expression. The shape is:

```
class Ring { public int[] Slots { get; } = new int[5];
}

// C# 13: ^ allowed inside the member's initializer block.
var r = new Ring { Slots = { [^1] = 99, [^2] = 42 } };
// sets Slots[4]=99, Slots[3]=42
```

`(new int[5] { [^1] = 99 })` does *not* compile — array-creation initializers take a positional element list, not indexed assignments.) This pairs naturally with code that then pattern-matches the result.

The interaction to internalize: a list pattern's correctness depends on `Length / Count` being a faithful, cheap, side-effect-free measure. For arrays, spans, `List<T>`, and strings this holds. For a custom collection where `Count` triggers enumeration or network I/O, a list pattern silently inherits that cost on

every match attempt — and because the length read happens before any element probe, even `[_, ..]` (which only cares that there's at least one element) pays the full `Count`. If “is there at least one element” is the question and `Count` is expensive, use `.Any()` /an enumerator, not a list pattern.

Performance: how the compiler lowers a switch

The Roslyn compiler does real work to make switches fast. Understanding the three lowering strategies tells you how to write switches the JIT will love:

1. **Jump table (`switch` IL opcode).** Emitted when arms test a *dense set of integer or enum constants* (and string switches above a threshold, via hashed dispatch). $O(1)$ dispatch regardless of arm count. The `Classify(int)` example's bare-constant arms `( 429 , 301 , 308 )` feed this.
2. **Balanced decision tree / binary search.** For sparse constants or relational ranges, the compiler builds a decision DAG that minimizes comparisons — it factors common sub-tests so that `>= 200 and < 300` and `>= 300 and < 400` share the single `< 300` boundary test rather than re-comparing. A well-ordered relational switch is typically $O(\log n)$ comparisons.
3. **Linear if-else chain.** The fallback for type tests (each `is T` is a separate `isinst / castclass`), `when` guards (always linear — a guard can fail, so the compiler must try the next arm), and patterns it can't reorder. Type-hierarchy switches are inherently linear in the number of distinct types tested, though the compiler reorders independent type tests and shares deconstruction calls where it can.

Two practical levers:

- **Order matters for the chain, not the table.** In a jump table, arm order is irrelevant to speed ($O(1)$ either way). In a linear chain, put the hot/common cases first.
- **Deconstruct once.** When several arms positional-match the same record, the compiler emits one `Deconstruct` call

and reuses the temporaries — but only if the arms are adjacent and structurally compatible. Interleaving a property-pattern arm between two positional arms over the same type can force a second deconstruction. Group like-shaped arms.

GC note: pure constant/relational/type/property patterns allocate nothing. The allocation sources are exactly: (a) a *binding* slice over an array (`GetSubArray` → new array), (b) `ToList` / `ToArray` you wrote to make a list pattern compile against `IEnumerable`, and (c) boxing when you pattern-match a value type through an `object`-typed input (the box happens at the call site, not in the pattern). Slice-binding over `Span<T>` / `ReadOnlySpan<T>` allocates nothing — which, combined with C# 14's implicit span conversions, makes span-based list patterns the allocation-free default for shape-matching hot paths.

Worked example: a tiny expression evaluator

The following ties together positional, list, constant, relational, `var`, and slice patterns, an exhaustive sealed-hierarchy switch, and a span-based tokenizer match — a realistic shape for an interpreter's core.

```

using System.Globalization;

abstract record Expr;
sealed record Num(double Value) : Expr;
sealed record Bin(char Op, Expr L, Expr R) : Expr;
sealed record Neg(Expr Operand) : Expr;

static class Calc
{
    // Deliberately NO `_` catch-all: omitting it is
    // what lets the compiler tell
    // us, by name, when a new Expr subtype goes
    // unhandled. `null` is matched
    // explicitly because it is always a possible
    // input. Even so, because `Expr`
    // is a *record* base (records expose a protected
    // copy constructor, so the
    // hierarchy isn't truly closed to the compiler),
    // Roslyn keeps emitting
    // CS8509; with CS8509 promoted to an error in the
    // .csproj, that promotion is
    // exactly the build-time gate we want – add a
    // subtype and the build breaks.
    public static double Eval(Expr e) => e switch
    {
        Num(var v) => v,
        Neg(var inner) => -
            Eval(inner),
        Bin('+', var l, var r) => Eval(l) +
            Eval(r),
        Bin('-', var l, var r) => Eval(l) -
            Eval(r),
        Bin('*', var l, var r) => Eval(l) *
            Eval(r),
        Bin('/', var l, var r) when Eval(r) is var d
            && d != 0 => Eval(l) /
                d,
        Bin('/', _, _) => throw new
            DivideByZeroException(),
        Bin(var op, _, _) => throw new
            NotSupportedException($"op '{op}'"),
        null => throw new
            ArgumentNullException(nameof(e)),
    };

    // List + slice patterns over a token array.
    // Implicit span conversion (C# 14)
    // lets a string[] flow into the
    // ReadOnlySpan<string> parameter with no
    // .AsSpan().
    public static Expr Parse(ReadOnlySpan<string>
        tokens) => tokens switch

```

```

{
    [var single] when double.TryParse(single,
        CultureInfo.InvariantCulture, out var n)
        => new Num(n),
    ["-", .. var rest]
        => new
        Neg(Parse(rest)),
    [var a, var op, .. var b] when op is "+" or "-"
        or "*" or "/"
        => new Bin(op[0],
        Parse([a]), Parse(b)),
    []
        => throw new
        FormatException("empty expression"),
    _
        => throw new
        FormatException($"cannot parse {tokens.Length}
        tokens"),
};
}

// Eval(Calc.Parse(["3", "+", "4", "*", "2"]))
// => Bin('+', Num 3, Bin('*', Num 4, Num 2)) -> 11
// (left-assoc, naive)
// Eval(Calc.Parse(["-", "5"])) =>
// -5
// Calc.Eval(new Bin('/', new Num(1), new Num(0)))
// throws DivideByZeroException

```

Points worth dwelling on. Every `Expr` subtype is matched, and the `Bin` cases end in a total `Bin(var op, _, _)` catch-all — drop any concrete `Bin` arm and that catch-all still covers it. There is deliberately **no `_` discard arm**: a discard would silently swallow any future subtype, defeating the entire reason to switch over a closed domain. With no discard, the compiler is forced to tell you, by name, what it can't account for. And it does have something to flag: because `Expr` is a *record* base (records expose a `protected` copy constructor, so the hierarchy is never truly closed to the compiler) plus a possible `null`, Roslyn cannot prove totality. We handle `null` explicitly; the residual openness still draws CS8509, and **that is the feature** — with CS8509 promoted to an error in the `.csproj`, the warning becomes a build gate. Add a new subtype (say `record Var(string Name) : Expr`) and the build breaks, with CS8509 naming the unhandled shape. (If you wanted the warning gone entirely rather than as a gate, you'd reach for a closed-hierarchy diagnostic suppressor or — once it ships — model the domain as a discriminated `union` type, whose patterns are exhaustiveness-

checkable by construction; that's a C# 15 feature, not available in the C# 14/.NET 10 baseline of this chapter. Either way, don't bolt on a `_` arm.) The `Bin('/', ...)` pair shows correct ordering: the guarded non-zero-divisor arm precedes the unguarded zero-divisor arm, so the specific success case wins before the throwing fallback — reverse them and the guard arm becomes unreachable for the divide-by-zero input but, worse, the unguarded `Bin('/', _, _)` would shadow the *valid* division too. The guard `Eval(r) is var d && d != 0` both binds `d` (avoiding a third `Eval(r)` call) and gates the arm; this is the idiomatic “compute-and-test in the pattern” trick.

In `Parse, ["-", .. var rest]` binds a slice — but over a `ReadOnlySpan<string>`, so `rest` is itself a `ReadOnlySpan<string>` carved out in $O(1)$ with **zero allocation**, and the recursive `Parse(rest)` flows it straight back in. Had `tokens` been `string[]`, every `.. var rest` would call `GetSubArray` and allocate a fresh array per recursion level. The collection expression `[a]` materializes a tiny single-element span/array for the recursive single-token parse; for a real parser you'd thread spans throughout to keep the whole parse allocation-free.

Choosing patterns deliberately

Patterns are not free of cost-of-readability. A switch with eight nested positional-and-property arms can be *less* legible than a handful of named helper predicates. Reach for the pattern grammar when (a) you are dispatching over a closed set of shapes and want compiler-enforced exhaustiveness, (b) you need to test-and-bind in one expression, or (c) you are shape-matching sequences where span-based list patterns give you allocation-free structural tests. Avoid it when a `when` guard would be doing all the real work (you've left the declarative world — write an `if`), when matching forces you to materialize a lazy sequence, or when a single property comparison reads more clearly as `x.Foo == y`. The compiler will reward well-ordered, guard-light, closed-domain switches with jump-table dispatch and dead-stripped throw guards; it cannot rescue a switch whose arms

overlap by accident or whose exhaustiveness depends on a run-time exception you forgot was there.

Sources: - [Pattern matching overview — Microsoft Learn](#) - [Patterns — C# reference, Microsoft Learn](#) - [Resolve pattern matching errors and warnings — Microsoft Learn](#) - [Implicit Span conversions in C# 14 — Start Debugging](#) - [The Ultimate Guide to List and Span Patterns in C# 14](#).

4 · Nullable Reference Types and Flow Analysis

Nullable reference types (NRT) are not a runtime feature — they are a *contract system* the C# compiler enforces at compile time by tracking, statement by statement, whether each expression *might* be null. This chapter dissects that flow analysis, the annotation grammar that feeds it, the

`System.Diagnostics.CodeAnalysis` attributes that let you teach the compiler facts it cannot infer, and the C# 14 additions — chiefly null-conditional assignment — that change behavior, not just syntax. The throughline: a nullable annotation is a *promise to your callers*, and the analyzer is the tool that checks you keep it.

The two contexts: annotation vs warning

NRT is governed by two independent switches the compiler tracks per-line. Conflating them is the single most common source of confusion.

- The **annotation context** decides whether the `?` suffix and the bare reference type carry meaning. When *enabled*, `string` means “I promise this is never null” and `string?` means “this may be null.” When *disabled*, both are *oblivious* — the type is neither nullable nor non-nullable, and the compiler makes no claims.
- The **warning context** decides whether the compiler *acts* on nullability mismatches by emitting `CS86xx` diagnostics.

You rarely set these independently in practice, but the distinction matters for migration. The project-wide setting in your `.csproj` collapses both:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <LangVersion>14</LangVersion>
  <TargetFramework>net10.0</TargetFramework>
</PropertyGroup>
```

`<Nullable>` accepts four values: `enable` (both on), `disable` (both off — the pre-C# 8 world), `warnings` (warning context on, annotation context off — diagnostics fire but `?` is ignored), and `annotations` (annotations meaningful, but no warnings emitted). The two narrow modes exist almost entirely for staged migration of a large codebase, which is where the `#nullable` directives earn their keep.

Migration strategy and annotation islands

For a legacy codebase, flipping `<Nullable>enable</Nullable>` globally can surface thousands of warnings at once. The disciplined approach is *file-at-a-time* using the `#nullable` directive, which overrides the project setting for a region of source:

```
#nullable enable           // turn both contexts on
                           from here
public sealed class Account
{
    public string Owner { get; }           //
                                         non-null promise
    public string? Nickname { get; set; }  //
                                         explicitly nullable

    public Account(string owner) => Owner = owner;
}
#nullable restore          // revert to the project-
                           level setting
```

`#nullable` takes `enable`, `disable`, `restore`, and an optional target — `#nullable enable annotations` or `#nullable disable warnings` — for the same four-way control at lexical granularity. The directive's scope is the source span from the directive to the next `#nullable` (or end of file); it never crosses file boundaries, so each file independently inherits the `.csproj` default until its first directive. `restore` resets to whatever the

`.csproj` dictates; reach for it after a narrow `#nullable disable` region (e.g. around generated or legacy code embedded mid-file) so the surrounding file’s context resumes rather than staying disabled to the file’s end.

The practical migration recipe: set the project to `<Nullable>enable</Nullable>`, then bulk-insert `#nullable disable` at the top of every not-yet-converted file. Convert files one at a time by deleting their `#nullable disable` line and fixing the warnings that appear. This gives you a shrinking, *visible* backlog (grep for `#nullable disable`) instead of an undifferentiated wall of diagnostics.

When you consume a library compiled *without* NRT — a “nullable-oblivious” assembly — its reference types are oblivious to you: the compiler neither trusts nor distrusts them, so calling `legacyApi.GetThing().Length` produces no warning even though `GetThing()` might return null. The inverse case, an **annotation island** — a small `#nullable enable` region inside an otherwise-disabled file — is the safe way to start: you get the analysis where you’ve reasoned about it without committing the whole file.

How flow analysis actually works

The compiler assigns every tracked expression one of two *null states* at each point in the control flow: **not-null** or **maybe-null**. (A third, internal *default* state exists for unassigned values but you rarely observe it directly.) State is not a property of the *type* — it is a property of the *expression at a program point*, and it flows through branches, assignments, and tests.


```

void Describe(string? text)
{
    // text: maybe-null here
    Console.WriteLine(text.Length); // CS8602:
    // dereference of a possibly null reference

    if (text is null)
        return;
    // text: not-null on this path (the maybe-null
    // path returned)

    Console.WriteLine(text.Length); // fine – no
    // warning

    text = null;
    // assignment resets state to maybe-null
    Console.WriteLine(text.Length); // CS8602 again
}

```

The analyzer understands a surprising range of tests as *null-state mutations*: `is null` / `is not null`, `== null` / `!= null` (including against custom `==` overloads, which it conservatively trusts), pattern matches (`is { } value`, `is string s`), `?.`, `??`, and `??=`. It also performs **definite-assignment-style** propagation across `&&`, `||`, the ternary, and early-exit control flow (`return`, `throw`, `continue`, `break`). What it does *not* do is interprocedural analysis: a helper method `bool IsValid(string? s)` that null-checks `s` internally teaches the caller nothing — the compiler sees only the signature. That gap is exactly what the nullability attributes close.

The flow analysis is a forward dataflow pass over the method’s control-flow graph, merging null-state at join points with a “worst case wins” rule: where two branches reconverge, an expression is not-null *only if* it was not-null on every incoming path. This is why a value proven non-null inside one arm of an `if` reverts to maybe-null after the `if` closes unless the other arm also established it (or exited). It also explains why `??` is so effective at clearing state — `value ?? throw new InvalidOperationException()` leaves `value`’s result not-null because the null path is provably gone. The analyzer additionally tracks *conditional* state across boolean-returning expressions:

after `bool ok = text is not null;`, testing `if (ok)` does **not** restore `text` to not-null, because the compiler doesn't track the dependency through an intermediate `bool` local. That asymmetry — direct tests narrow, indirected ones don't — is the motivation for the `NotNullWhen` family of attributes, which re-establish the link a `bool` return would otherwise sever.

One more rule shapes everyday code: the analyzer treats `throw` expressions and `[DoesNotReturn]` calls as flow terminators, so the *unreachable* code after them inherits no pessimism. This lets you front-load validation as guard clauses and have the rest of the method see clean, narrowed state — the structural pattern modern C# pushes you toward, and the reason “fail fast at the top” and “good null-state downstream” are the same habit.

A subtle but important rule: the analyzer's state for a field or property is *invalidated by any method call* it cannot prove is pure, because the call might mutate the member. Capturing into a local before testing is the canonical fix:

```
string? cached = _nickname;
if (cached is not null)
    Use(cached);           // local can't be mutated
                           by Use; state survives
```

Nullability attributes: teaching the compiler what it cannot infer

The attributes live in `System.Diagnostics.CodeAnalysis` and are *consumed* by the compiler — they emit no IL and have no runtime effect. They fall into three groups: **preconditions** (`AllowNull`, `DisallowNull`) that loosen or tighten what callers may pass; **postconditions** (`MaybeNull`, `NotNull`, `MaybeNullWhen`, `NotNullWhen`, `NotNullIfNotNull`) that describe outputs; and **flow** attributes (`DoesNotReturn`, `DoesNotReturnIf`, `MemberNotNull`, `MemberNotNullWhen`) that shape control-flow and member-initialization analysis. All were introduced with the NRT feature in C# 8 / .NET Core 3.0 and are unchanged in behavior through .NET 10.

The two preconditions are easy to overlook but solve real problems. `[AllowNull]` on a property *setter* or parameter says “a non-nullable-typed member may nonetheless accept null from callers” — the canonical example is a property typed `string` that returns a default when never set but tolerates `someObj.Name = null;` to reset. `[DisallowNull]` is the mirror: a property typed `string?` whose *getter* may return null but whose setter refuses it. These let the *read* and *write* nullability of a single member diverge, which the bare `?` / non-`?` type cannot express. Reach for them only when input and output nullability genuinely differ; for the common case where they agree, the plain annotation is clearer. The remainder of this section drills into the five attributes you will actually reach for weekly.

NotNullWhen — the TryGetValue pattern

The classic `Try` pattern returns `bool` and yields a value through an `out` parameter that is meaningful *only* when the method returns `true`. Without help, the compiler must assume the `out` is maybe-null on every path. `[NotNullWhen(true)]` ties the two together:

```

using System.Diagnostics.CodeAnalysis;

public static class Headers
{
    public static bool TryGetContentType(
        IReadOnlyDictionary<string, string> headers,
        [NotNullWhen(true)] out string? value)
    {
        // Declare 'found' non-nullable: TryGetValue
        // carries [MaybeNullWhen(false)],
        // which does NOT narrow a 'string?'-declared
        // out to not-null on the true
        // side of '&&'. A non-nullable 'out string
        // found' makes the inner
        // dereference warning-free.
        if (headers.TryGetValue("Content-Type", out
            string found) && found.Length > 0)
        {
            value = found;
            return true;
        }
        value = null;
        return false;
    }
}

// Consumption under nullable analysis:
if (Headers.TryGetContentType(request.Headers, out
    string? ct))
{
    // ct: not-null here because the method returned
    // true
    Console.WriteLine(ct.ToUpperInvariant()); // no
    CS8602
}
else
{
    // ct: maybe-null on the false branch
    Console.WriteLine(ct?.Length ?? 0); // ?
    // required, correctly
}

```

`int.TryParse` is annotated exactly this way —

`[NotNullWhen(true)] out int result` would be meaningless for a value type, so it simply uses `out int result` and the bool return narrows nothing, but reference-typed `Try` methods such as `Uri.TryCreate(string?, ..., [NotNullWhen(true)] out Uri? result)` carry `NotNullWhen(true)` so the parsed value is

not-null on success. (`Dictionary<TKey, TValue>.TryGetValue` is the generic cousin and uses `[MaybeNullWhen(false)]` instead — see the worked example for why.) This is why consuming these “just works” under NRT.

NotNullIfNotNull — threading nullability through a transform

When a method’s output nullability *mirrors* an input — null in, null out; non-null in, non-null out — encode that with `[return: NotNullIfNotNull(nameof(parameter))]`. Using `nameof` (rather than a string literal, the pre-C# 11 style) keeps it refactor-safe:

```
using System.Diagnostics.CodeAnalysis;

public static class PathHelpers
{
    // If 'input' is non-null, the result is non-null;
    // if null, the result is null.
    [return: NotNullIfNotNull(nameof(input))]
    public static string? NormalizeSlashes(string?
        input)
    {
        if (input is null)
            return null;
        return input.Replace('\\', '/');
    }
}

// Caller-side flow:
string? maybe = GetUserInput();           // maybe-
                                           null
string? a = PathHelpers.NormalizeSlashes(maybe);
//    a: maybe-null → matches input

string concrete = "C:\\temp";              // not-null
string b = PathHelpers.NormalizeSlashes(concrete);
//    b: not-null → compiler lets you assign to a
//              non-nullable local with no '!'
Console.WriteLine(b.Length);              // no
                                           warning
```

The payoff is in the second call: because `concrete` is provably not-null, the result is treated as not-null and assigns to a non-nullable `string` with no null-forgiving operator. The compiler

threaded the contract for you. `Path.GetExtension`, `Path.ChangeExtension`, and many string helpers carry exactly this annotation in .NET 10.

`MemberNotNull` — trusting a field after `Initialize()`

A common structural pattern: a field is not assigned in the constructor but in an `Initialize()` / `EnsureLoaded()` method called before use. The compiler can't know `Initialize()` guarantees the field, so it warns at every later dereference.

`[MemberNotNull(...)]` declares the postcondition that the listed members are non-null when the method returns normally:

```

using System.Diagnostics.CodeAnalysis;

public sealed class ConfigStore
{
    private Dictionary<string, string>? _entries; // not set in ctor
    private string? _source;

    public ConfigStore() { } // no CS8618 because the fields are nullable

    [MemberNotNull(nameof(_entries), nameof(_source))]
    public void Load(string path)
    {
        _source = path;
        _entries = File.ReadLines(path)
            .Select(l => l.Split('=', 2))
            .Where(p => p.Length == 2)
            .ToDictionary(p => p[0].Trim(),
                p => p[1].Trim());
        // Reaching the end normally means BOTH fields are non-null –
        // exactly what the attribute promises. If a code path could leave
        // _entries null, the compiler warns HERE (CS8774), not at the caller.
    }

    public string Get(string key)
    {
        Load(_source ?? "default.cfg");
        // After Load() returns, _entries is trusted non-null:
        return _entries[key]; // no CS8602
    }
}

```

`MemberNotNull` is verified at the *definition* site: if any normal-return path through `Load` leaves `_entries` possibly null, you get `CS8774` on `Load` itself. That's the right place for the error — it keeps the promise honest. The conditional cousin `[MemberNotNullWhen(true, nameof(_entries))]` is for `bool`-returning probes like `TryLoad`.

DoesNotReturn — guard clauses that prune flow

A throw-helper annotated `[DoesNotReturn]` tells the analyzer that code after a call to it is unreachable, so any null-state pes-

simism is discarded on that path. This is how you keep guard helpers from defeating flow analysis:

```
using System.Diagnostics.CodeAnalysis;

internal static class Guard
{
    [DoesNotReturn]
    public static void ThrowArgNull(string name) =>
        throw new ArgumentNullException(name);
}

public static int Length(string? value)
{
    if (value is null)
        Guard.ThrowArgNull(nameof(value));    //
        analyzer: this path terminates

    return value.Length;                        // value:
        not-null – no CS8602
}
```

Without `[DoesNotReturn]`, the compiler assumes `Guard.ThrowArgNull` *might* return, so `value` stays maybe-null and `value.Length` warns. The BCL's own `ArgumentNullException.ThrowIfNull` (introduced in .NET 6) combines this with a `[NotNull]` parameter annotation, which is why a single `ArgumentNullException.ThrowIfNull(value);` line both throws *and* upgrades `value` to not-null downstream — prefer it over hand-rolled guards in new code.

The null-forgiving operator `!` — and when it's legitimate

The postfix `!` (the *null-forgiving* or *dammit* operator, C# 8) overrides the analyzer: `expr!` is treated as not-null regardless of its inferred state. It generates *no IL* — it is purely a compile-time assertion that you, the author, know better than the flow analysis.

It is legitimate in a narrow set of cases:

1. **Facts the analyzer cannot express.** After

`Debug.Assert(x is not null)` in a build configuration where asserts are stripped, or when an invariant is enforced by a framework the compiler can't see:

```
// EF Core / serializers set this via reflection
// after construction.
public string Name { get; set; } = null!; //
// legitimate: framework guarantees it
```

2. **Test code** asserting a value is present:

```
result!.Value.Should().Be(...).
```

3. **Interop with oblivious or mis-annotated libraries**, where you've verified the real contract.

It is *illegitimate* — and the dominant antipattern — when used to silence a warning instead of fixing the contract. Writing `customer.Order!.Total` because the compiler complained, without establishing *why* `Order` is non-null here, converts a compile-time warning into a latent `NullReferenceException`. Each `!` is a place where you've turned off the safety system; they should be rare, commented, and auditable (a `grep -c '!\\.'` count is a crude but useful code-smell metric). Reach for `!` only after asking “can I express this with an attribute or a null check instead?” — usually you can.

Generics and nullability

Generics are where NRT gets genuinely subtle, because an unconstrained `T` can be substituted with *either* a reference type *or* a value type, and `T?` means different things in each case.

For an unconstrained `T`, `T?` denotes a “maybe-default” value — it does **not** add `Nullable<T>` for value types. With `T == int`, `T?` is still `int` (whose default `0` is the “default” sentinel); with `T == string`, `T?` is `string?`. The annotation expresses *intent* (this may be the default/null) more than a distinct type. This duality is why `Nullable<T>` and `T?`-on-a-reference are not interchangeable concepts despite sharing syntax: the former is a distinct value-type wrapper (`System.Nullable<T>`) with its own

layout and `.HasValue` / `.Value` members, instantiated per value-type argument; the latter is pure compile-time metadata over the same runtime reference type. When you constrain `where T : struct`, `T?` does become `Nullable<T>` and gains `.HasValue` / `.Value`; when you constrain `where T : class`, `T?` is a nullable reference and stays the same runtime type. The unconstrained middle ground is the only place the “maybe-default” reading applies, and it is precisely the case the analyzer cannot fully resolve until instantiation — so it errs toward letting `default` flow where a constrained version would warn.

The infamous **default-constraint problem**: returning `default(T)` from a method declared to return `T` warns, because for reference types `default` is null:

```
public static T First<T>(IEnumerable<T> source)
{
    foreach (T item in source)
        return item;
    return default; // CS8603 for reference T:
                   // returning maybe-null as T
}
```

The fix is to make the *signature honest* — declare the return as `T?` and let callers handle the maybe-default:

```
public static T? FirstOrDefault<T>(IEnumerable<T>
    source)
{
    foreach (T item in source)
        return item;
    return default; // now consistent with T? return
                   // - no warning
}
```

When you genuinely require a *non-null reference type* or *any value type* as the key — the classic dictionary scenario — use the `notnull` constraint (C# 8). It forbids both nullable reference types and nullable value types as the type argument.

Worked example: a generic `Cache<TKey, TValue>` done correctly

This ties the chapter together: the `NotNull` constraint on the key, a `TryGet` whose `out` carries `MaybeNullWhen(false)` — the BCL’s idiom for a generic value that may be a value *or* reference type — honest signatures throughout, and a `GetOrAdd` that returns a real value without a single `!`.

```

using System.Diagnostics.CodeAnalysis;

public sealed class Cache<TKey, TValue> where TKey :
    notnull
{
    private readonly Dictionary<TKey, TValue> _map =
        new();
    private readonly Func<TKey, TValue> _factory;

    public Cache(Func<TKey, TValue> factory) =>
        _factory = factory;

    // MaybeNullWhen(false): the out MAY be the
        default on the false branch.
    // This is the right attribute for an
        unconstrained generic out, because
    // the parameter type 'TValue' is non-nullable but
        could be substituted
    // with a value type – where [NotNullWhen(true)]
        (which requires a nullable
    // parameter type) would not apply. It forwards
        Dictionary.TryGetValue's
    // own annotation unchanged.
    public bool TryGet(TKey key,
        [MaybeNullWhen(false)] out TValue value) =>
        _map.TryGetValue(key, out value);

    public TValue GetOrAdd(TKey key)
    {
        // Declare the out as the non-nullable
            'TValue' – matching _map's value
        // type. The compiler does not "narrow to not-
            null on true"; rather, the
        // declared non-nullable type means
            dereferencing/returning 'existing'
        // produces no warning, and on the true branch
            a stored value really is
        // present, so the return is honest.
        if (_map.TryGetValue(key, out TValue
            existing))
            return existing; // no CS8603 –
            'existing' is typed non-nullable

        TValue created = _factory(key);
        _map[key] = created;
        return created;
    }

    public void Remove(TKey key) => _map.Remove(key);
}

```

```
// Usage – TKey constraint rejects nullable keys at
// compile time:
var lengths = new Cache<string, int>(s => s.Length);
int n = lengths.GetOrAdd("hello"); //
// 5

var profiles = new Cache<int, Profile>(id =>
    Profile.Load(id));
// Declare the out non-nullable.
// [MaybeNullWhen(false)] makes 'p' maybe-null
// only where TryGet returned false; inside the true
// branch it is treated as a
// real (non-nullable) value, so the dereference is
// warning-free.
if (profiles.TryGet(42, out Profile p))
    Console.WriteLine(p.DisplayName); //
// no CS8602

// Compile error – string? violates 'where TKey :
// notnull':
// var bad = new Cache<string?, int>(s => 0); //
// CS8714
```

`[MaybeNullWhen(false)]` is what `Dictionary.TryGetValue` itself carries on `.NET 10` (`out TValue value`): it says “the out may be the default when I return false,” which is the honest contract for a lookup. It is the *non-nullable-parameter* counterpart to `[NotNullWhen(true)]`, not an exact dual — they apply to different parameter types and narrow differently.

`NotNullWhen(true)` requires a *nullable* parameter type (`out string?`) and upgrades a nullable out to not-null on the true branch; `MaybeNullWhen(false)` sits on a *non-nullable* parameter type (`out TValue`) and only weakens it to maybe-null on the false branch. The BCL deliberately chose

`MaybeNullWhen(false)` for generic `Try` methods because `TValue` could be substituted with a value type, where `out TValue?` would change meaning to `Nullable<T>`. A consequence worth internalizing: `MaybeNullWhen(false)` does **not** narrow a *nullable-declared* out (`out Profile? p`) to not-null on the true branch — only a non-nullable declaration (`out Profile p`) gives you a warning-free dereference there. Using `notnull` on `TKey` mirrors `Dictionary<TKey, TValue>`’s own constraint and prevents a whole class of dictionary-key bugs at the call site (`CS8714`).

Null-conditional assignment (C# 14)

Introduced in C# 14 (.NET 10), assignment is now permitted with a null-conditional access on the left-hand side: `x?.Prop = value` and `x?[i] = value`. Before C# 14 this was a compile error; you had to write an explicit `if`. The feature is more than sugar — its evaluation semantics differ from the naive expansion in a way you must internalize.

The compiler-defined semantics: `P?.A = B` is equivalent to `if (P is not null) P.A = B;`, with `P` evaluated exactly once. The load-bearing consequence — and the chapter's headline gotcha — is that **the right-hand side `B` is evaluated only when `P` is non-null**. The assignment short-circuits the *entire* right side, not just the member access.

```
public sealed class Customer
{
    public Order? Order { get; set; }
}

Customer? customer = LookupCustomer(id);    // may be
null

// C# 14: if customer is null, ComputeOrder() is NEVER
// called.
customer?.Order = ComputeOrder();

Order ComputeOrder()
{
    Console.WriteLine("computing...");    // side
    effect - does NOT run when customer is null
    return new Order { Total = 100m };
}
```

If `customer` is null, you will not see `"computing..."` printed. The expression is equivalent to:

```
if (customer is not null)
    customer.Order = ComputeOrder();
```

Contrast this with what many developers *assume* is equivalent — and what the old workaround often looked like:

```
// NOT equivalent: ComputeOrder() runs
// unconditionally, then the result is
// thrown away if customer is null. Different side-
// effect profile, possibly
// different cost (allocations, I/O, logging).
Order tmp = ComputeOrder();
if (customer is not null)
    customer.Order = tmp;
```

This is why the feature is a *behavior change*, not pure syntactic sugar: code that relied on the right-hand side running for its side effects will behave differently if mechanically rewritten into the new form. The indexer form behaves identically:

```
string[]? buffer = TryRentBuffer();
buffer?[0] = ComputeFirst(); // ComputeFirst()
// called only if buffer is non-null
```

The receiver `P` is still evaluated once even when it is a complex expression, matching the long-standing single-evaluation guarantee of `?.` reads. The left side may be a property, indexer, or field — and, notably, an *event* via `+= / -=`: conditionally attaching a handler with `control?.Click += OnClick;` is one of the feature’s two headline motivations (the other is property/ `Set()` parity). What is *not* allowed: ref-assignment to a conditional target (`a?.b = ref x;`) and deconstruction assignment (`(a?.b, c?.d) = (x, y);`). The receiver of the conditional access must be a *reference* type: `a?.b = c` is disallowed when `a` is a non-nullable value type — it can never be null, so the conditional is meaningless — and is likewise rejected for a `MyStruct? a`, because `a.Value` is not a variable and so has no assignable target. Null-conditional assignment composes with compound assignment (`+=` , `-=` , `??=` , ...) in C# 14 as well — `customer?.Order ??= ComputeOrder()` and `counter?[key] += 1` follow the same “evaluate right side only if receiver non-null” rule. Increment and decrement (`++ / --`) are *not* permitted on a null-conditional target (`a?.b++;` is an error).

There is also a flow-analysis interaction worth knowing: after `customer?.Order = ComputeOrder();` the compiler does **not**

treat `customer` as non-null on subsequent lines, because the statement completes whether or not the assignment happened. If you need `customer` narrowed afterward, the null-conditional form is the wrong tool — use an explicit `if (customer is not null) { ... }` block so the narrowed state survives into the body. The null-conditional assignment is a *fire-and-forget* convenience for the “do it only if present, and I don’t care to know which” case.

Use it to flatten the very common “set this property if the optional object exists” pattern, especially in builders and configuration code:

```
// Before C# 14:  
if (options.Logging is not null)  
    options.Logging.MinimumLevel = LogLevel.Debug;  
  
// C# 14:  
options.Logging?.MinimumLevel = LogLevel.Debug;
```

Do **not** use it where the right-hand side *must* execute regardless (e.g., it advances a sequence, logs, or has transactional side effects). In that case the short-circuit is a bug, and the explicit two-line form documents intent better.

Designing public APIs that communicate nullability honestly

The discipline NRT imposes pays off most at *API boundaries*, where your signature is the contract a thousand callers read.

Principles:

- **Return non-null collections, not null.**

`ReadOnlyList<T>` that is empty beats

`ReadOnlyList<T>?` that is sometimes null. Callers should never need a null check to iterate.

- **Use `?` to mean “absence is normal,”** and reserve a thrown exception for “this should never be absent.”

`TryGet` -with- `[NotNullWhen(true)]` is the right shape

when absence is common; a non-nullable return that throws is right when absence is exceptional.

- **Annotate outputs precisely** with `NotNullIfNotNull` / `MaybeNullWhen` so callers inherit accurate state instead of having to `!` their way past your imprecision.
- **Constrain generics** (`notnull`, or `where T : class` / `where T : struct`) when the nullability story differs by category — it's part of the signature's honesty.
- **Never use `!` in a public surface to paper over a contract you haven't actually established.** If `Name` can be null, type it `string?`; don't expose `string` and `!`-assign internally and hope.

A signature read in isolation should answer “can this be null?” without consulting documentation. That is the whole point.

Pitfalls recap

- **`!` as a warning-silencer.** Each `!` disables the safety system at one point. Audit them; prefer attributes or real null checks. A spreading population of `expr!` means the team is fighting the analyzer instead of using it.
- **Assuming runtime enforcement.** Annotations and attributes emit no IL and perform no runtime checks. A `string` parameter *can* receive null at runtime — from oblivious callers, reflection, deserializers, or `default`. For genuinely public, security-relevant entry points, still call `ArgumentNullException.ThrowIfNull` (introduced .NET 6). NRT is a *design-time* contract, not a runtime guard.
- **`x?.Prop = value` does not evaluate the right side when `x` is null (C# 14).** This is the behavior change, not just new syntax. Do not mechanically migrate code whose right-hand side has required side effects.
- **State invalidated by method calls.** After a non-pure call, the analyzer re-pessimizes field/property null-state. Capture into a local before testing.

- **default(T) for unconstrained T**. It's null for reference types; declare the return `T?` rather than reaching for `!`.

Performance and runtime notes

There is essentially nothing to measure: NRT is a compile-time-only feature. The annotations (`?`, the attributes) are recorded as metadata —

`System.Runtime.CompilerServices.NullableAttribute` and `NullableContextAttribute` are emitted into the assembly so *consumers* get the same flow analysis — but they generate no executable code, no allocations, and no JIT or GC impact. The null-forgiving `!` likewise compiles to nothing. Null-conditional assignment (C# 14) lowers to the same branch you'd write by hand (`if (P is not null) P.A = B;` with a single evaluation of `P`), so it is exactly as fast as the explicit form and never slower. The only runtime cost in this entire chapter is the one you *opt into* — calling `ArgumentNullException.ThrowIfNull`, which is a single non-null branch plus a throw helper, negligible on any hot path and worth it at trust boundaries.

The mental model to carry forward: nullable reference types turn a class of `NullReferenceException`s into compiler diagnostics *if and only if* your signatures and attributes describe reality. The analyzer is mechanical and honest; your job is to give it true premises.

Sources: - [Null conditional assignment — C# 14 feature specification \(Microsoft Learn\)](#) - [What's new in C# 14 \(Microsoft Learn\)](#) - [Attributes for nullable static analysis \(Microsoft Learn\)](#) - [Nullable reference types \(Microsoft Learn\)](#) - [NotNullIfNotNullAttribute \(.NET 10 API reference\)](#) - [MemberNotNullWhenAttribute \(.NET 10 API reference\)](#)

5 · Strings, Text, and Interpolation

Text handling sits at the boundary between programmer convenience and machine reality: a `string` is an ergonomic, immutable, UTF-16 sequence, but Unicode, allocation pressure, and culture-sensitive comparison all leak through that abstraction the moment you do anything serious. This chapter works from the syntax you write (raw and interpolated literals) down through the lowering the compiler performs, the BCL types that let you avoid allocations entirely, and the correctness traps — surrogate pairs, grapheme clusters, ordinal-vs-culture comparison — that separate code that *works on your machine* from code that works in Istanbul.

Raw string literals

Raw string literals arrived in **C# 11 (.NET 7)** and are the first feature to reach for whenever a string contains quotes, backslashes, or multiple lines. A raw string literal is delimited by *at least three* double-quote characters. Inside the delimiters nothing is an escape sequence — `\n`, `\t`, and `é` are all taken literally — so JSON, regex, file paths, and SQL stop fighting you.

```
string json = """{"name":"Ada","langs":  
    ["C#","F#"],"path":"C:\\temp\\x"}""";  
string regex = """\\d{3}-\\d{2}-\\d{4}""";    // no  
    doubled backslashes
```

The “at least three” rule exists so the content can itself contain quotes. The compiler counts the opening fence and requires the *same* count to close; any shorter run of quotes inside is content. If your text contains a run of three quotes, open with four:

```
string contains3Quotes = """She said """hello"""  
    loudly.""";
```

The multiline form is where raw strings earn their keep, and it has one rule that trips up everyone the first time. When the

opening `"""` is followed by a newline and the closing `"""` sits on its own line, the literal spans the lines *between* them, and **the whitespace to the left of the closing delimiter is stripped from every line**. The closing fence's indentation defines column zero:

```
string sql = """
    SELECT id, name
    FROM   users
    WHERE  active = 1
    """;
// Each line begins at "SELECT"/"FROM"/"WHERE"; the 4-
// space indent is removed.
```

This is the single most common raw-string pitfall. If the closing `"""` is indented *less* than the content, the extra content indentation is preserved (often not what you want); if a content line is indented *less* than the closing delimiter, you get a compile error (`CS8999` / `CS9000` -class diagnostics) because the line cannot have its prefix stripped. The mental model that keeps you sane: **align the closing fence with where you want the left margin to be**, not with the opening fence. Also note the opening newline and the final newline before the closing fence are not part of the value — `sql` above has no leading or trailing blank line.

Interpolated raw strings

Raw and interpolated literals compose. You prefix the opening fence with one or more `$` characters, and the **number of dollar signs sets how many braces are needed to open an interpolation hole**. This is the brace analogue of the quote-count rule and is what makes raw interpolation safe inside brace-heavy text like JSON or CSS:

```

int port = 5432;
string host = "db.internal";

// Single $: one brace opens a hole, but a literal {
//           needs no escaping unless it's a hole.
string one = $""
    Listening on {host}:{port}
    "";

// Double $$: TWO braces open a hole, so single braces
//           are literal JSON.
string config = $$""
    {
        "host": "{{host}}",
        "port": {{port}},
        "tls":  { "enabled": true }
    }
    "";

```

With `$$`, `{{host}}` is the interpolation and the bare `{ / }` of the JSON object are literal — no escaping, no doubling. This is the idiomatic way to template JSON, GraphQL, or Razor-like content in C#. Choose the dollar count to be one greater than the longest run of literal braces you need adjacent to a hole.

UTF-8 string literals

Also from **C# 11 (.NET 7)**, the `u8` suffix produces a `ReadOnlySpan<byte>` containing the UTF-8 encoding of the literal — *not* a `string`, and not UTF-16:

```

ReadOnlySpan<byte> crlf = "\r\n"u8;           // 2
                        bytes
ReadOnlySpan<byte> hello = "Héllö"u8;         // 'é'
                        is 2 UTF-8 bytes -> 6 bytes total

```

The bytes are embedded in the assembly's data section, so there is **zero runtime encoding work and zero heap allocation** — the span points directly at the loaded image. This matters precisely where text meets the wire or the disk: HTTP verbs and headers, protocol tokens, JSON property names, file signatures. The classic use is comparing an incoming byte buffer against a known token without first decoding it to a UTF-16 `string`:

```

static bool IsJsonContentType(ReadOnlySpan<byte>
    headerValue) =>
    headerValue.StartsWith("application/json"u8);

static bool LooksLikePng(ReadOnlySpan<byte> file) =>
    file.StartsWith([0x89, (byte)'P', (byte)'N',
        (byte)'G']) ||
    file.StartsWith("\x89PNG"u8); // same 4 bytes,
        expressed as a literal

```

When *not* to use `u8` : it is `ReadOnlySpan<byte>`, a ref struct, so you cannot store it in a field, box it, put it in an `async` local that crosses an `await`, or use it as a generic type argument. If you need a persisted UTF-8 blob, materialize it once into a `static readonly byte[]` (the `u8` span gives you the source: `"..."u8.ToArray()`). And remember the comparison is *byte* equality — it is case-sensitive and encoding-exact, which is exactly right for protocol tokens but wrong for human text.

How string interpolation lowers

An interpolated `string` looks like sugar for `string.Format`, but since **C# 10 (.NET 6)** the compiler lowers it through `DefaultInterpolatedStringHandler` (in `System.Runtime.CompilerServices`). Understanding this lowering is the key to both performance and to writing your own handlers. Given:

```

string msg = $"Order {id} total {amount:C2} at
    {when:t}";

```

the compiler emits roughly:

```
var handler = new DefaultInterpolatedStringHandler(
    literalLength: 17, // "Order " (6) + " total "
                      (7) + " at " (4)
    formattedCount: 3); // number of holes
handler.AppendLiteral("Order ");
handler.AppendFormatted(id);
handler.AppendLiteral(" total ");
handler.AppendFormatted(amount, format: "C2");
handler.AppendLiteral(" at ");
handler.AppendFormatted(when, format: "t");
string msg = handler.ToStringAndClear();
```

The handler rents a buffer (often stack/ `ArrayPool` -backed for the default handler), and the strongly-typed

`AppendFormatted<T>` overloads call

`ISpanFormattable.TryFormat` when `T` supports it — so an `int` or `decimal` is formatted directly into the buffer with **no intermediate `ToString()` allocation and no boxing**. That is why `$"x = {count}"` is faster and allocates less than the old `string.Format("x = {0}", count)`: the value type never boxes. The `literalLength` / `formattedCount` constructor arguments let the handler size its buffer up front.

One consequence worth internalizing: an interpolated string is only a `string` when the target type is `string` (or `object` / `IFormattable`). When the target type is something that *accepts an interpolated string handler* — like `StringBuilder.Append`, `Debug.Assert`, or your own API — the compiler binds to that handler instead and may skip the work entirely. That redirection is the whole point of custom handlers.

Writing a custom interpolated string handler

The headline scenario, also **C# 10**, is conditional logging: if the log level is disabled, you want to pay *nothing* — not the formatting, not even the boxing of the arguments. A custom `[InterpolatedStringHandler]` lets the compiler guard the entire `Append*` sequence behind a runtime check.

The contract: a handler is a (usually `ref struct`) type marked `[InterpolatedStringHandler]` whose constructor takes `int literalLength, int formattedCount` first. To receive *other arguments from the call site* — the logger, the level — you add parameters and tell the compiler to forward them with `[InterpolatedStringHandlerArgument]` on the method parameter. The magic that enables skipping is the **trailing out bool constructor parameter**: when the compiler sees it, it generates `if (handlerWasBuilt) { ...Append... }`, so a `false` short-circuits *before any `AppendFormatted` runs*.


```

using System.Runtime.CompilerServices;

public enum LogLevel { Trace, Debug, Info, Warn, Error
}

public sealed class Logger
{
    public LogLevel MinLevel { get; init; } =
        LogLevel.Info;

    public bool IsEnabled(LogLevel level) => level >=
        MinLevel;

    // The handler parameter is annotated so the
        compiler forwards `this` and `level`
    // into the handler's constructor.
    public void Log(
        LogLevel level,
        [InterpolatedStringHandlerArgument("",
            nameof(level))]
        LogInterpolatedStringHandler handler)
    {
        // If the handler short-circuited, IsEnabled
        is false and `handler` is empty.
        if (handler.IsEnabled)
            WriteLine(handler.GetFormattedText());
    }

    private static void WriteLine(string s) =>
        Console.WriteLine(s);
}

[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    private DefaultInterpolatedStringHandler _inner;
    public bool IsEnabled { get; }

    // literalLength + formattedCount are mandatory
        and first.
    // logger and level are forwarded from the call
        site.
    // The trailing `out bool shouldAppend` gates the
        Append* calls.
    public LogInterpolatedStringHandler(
        int literalLength,
        int formattedCount,
        Logger logger,
        LogLevel level,
        out bool shouldAppend)
    {

```

```

        isEnabled = logger.IsEnabled(level);
        shouldAppend = isEnabled;
        _inner = isEnabled
            ? new
                DefaultInterpolatedStringHandler(literalLength
                    , formattedCount)
                : default;
    }

    public void AppendLiteral(string s) =>
        _inner.AppendLiteral(s);

    public void AppendFormatted<T>(T value) =>
        _inner.AppendFormatted(value);

    public void AppendFormatted<T>(T value, string?
        format) =>
        _inner.AppendFormatted(value, format);

    public string GetFormattedText() =>
        _inner.ToStringAndClear();
}

```

The empty string `""` in

```
[InterpolatedStringHandlerArgument("", nameof(level))]
```

is the convention for “forward the receiver (`this`)”;

`nameof(level)` forwards the `level` parameter. **Order matters:** the names must appear in the same order as the corresponding constructor parameters after

`literalLength / formattedCount` . Now the call site:

```

var log = new Logger { MinLevel = LogLevel.Warn };
ExpensiveStats? stats = null;

// Info < Warn, so this is disabled: GetStats() is
// NEVER called, nothing is formatted.
log.Log(LogLevel.Info, $"Computed {ComputeExpensive()}
    from {stats!.Describe()}");

```

Because `shouldAppend` came back `false` , the compiler skipped the whole `Append*` block — including the calls to `ComputeExpensive()` and `stats.Describe()` . That is strictly better than `if (log.IsEnabled(...)) log.Log(...)` because the guard is implicit and the argument expressions are not evaluated. **When not to bother:** if every argument is a cheap field

and the level is usually enabled, a plain handler-free overload is simpler; the custom handler pays off when arguments are expensive or logging is mostly off in production.

CompositeFormat: caching parsed format strings

`string.Format("{0} of {1}", a, b)` re-parses that format string *every call*. When a format string is fixed but used in a hot loop, **.NET 8** introduced `System.Text.CompositeFormat` to parse once and reuse. Cache the parsed instance in a `static readonly` field and pass it to the `string.Format` / `StringBuilder.AppendFormat` overloads that accept a `CompositeFormat`:

```
using System.Text;

internal static class Messages
{
    // Parsed exactly once at type init; reused on
    // every Format call.
    private static readonly CompositeFormat
        ItemProgress =
            CompositeFormat.Parse("Processing item {0} of
                                  {1} ({2:P0})");

    public static string Progress(int done, int total)
        =>
            string.Format(

                System.Globalization.CultureInfo.InvariantCulture,
                ItemProgress, done, total, (double)done /
                total);
}
```

The analyzer `CA1863` will flag literal format strings used with `string.Format` in hot paths and suggest this. Note the tradeoff: `CompositeFormat` only helps the *parsing* cost, not the value-formatting cost, and an interpolated string handler already avoids re-parsing entirely (the literal/hole structure is baked into the IL). So reach for `CompositeFormat` when the format string is **chosen at runtime** (e.g., localized resources, a tem-

plate loaded from config) and reused — that is the case interpolation cannot cover, because interpolation requires a compile-time literal.

string.Create: building strings without intermediate allocation

When you know the final length and can fill the characters yourself, `string.Create<TState>(int length, TState state, SpanAction<char, TState>)` allocates the string *once* and hands you a mutable `Span<char>` over its storage. There is no `StringBuilder`, no `char[]`, no second copy. This is the right tool for fixed-width identifiers, formatted codes, and padding:

```
public static string FixedWidthId(string prefix, int
    number, int width)
{
    // total length is known up front
    int len = prefix.Length + width;
    return string.Create(len, (prefix, number, width),
        static (span, state) =>
    {
        var (p, n, w) = state;
        p.CopyTo(span);
        string.CopyTo(span) - no allocation
        Span<char> digits = span[p.Length..];
        if (!n.TryFormat(digits, out int written)) //
            ISpanFormattable, no allocation
            throw new ArgumentException("number too
                wide", nameof(number));
        // left-pad with '0' by shifting and filling
        int pad = w - written;
        if (pad < 0) throw new
            ArgumentException("number too wide",
                nameof(number));
        digits[..written].CopyTo(digits[pad..]);
        digits[..pad].Fill('0');
    });
}

// FixedWidthId("INV-", 42, 6) => "INV-000042"
```

A few expert notes. The `state` tuple is passed by the runtime, which is why we capture nothing in the lambda — mark it `static` so the compiler enforces no closure allocation.

`int.TryFormat` is the `ISpanFormattable` path: it writes digits straight into the span. We copy the prefix with `string.CopyTo(Span<char>)` (an instance method present since .NET Core 2.1) rather than `prefix.AsSpan().CopyTo(span)`; both are allocation-free, the former is just less ceremony. (Separately, the new **C# 14** implicit `string` $\rightarrow$ `ReadOnlySpan<char>` conversion — covered below — lets a `string` flow into any `ReadOnlySpan<char>` parameter without `.AsSpan()`; it does not change `CopyTo`, whose destination is a mutable `Span<char>`.) **When not to use** `string.Create`: if you don't know the length up front, or the fill logic is branchy and error-prone, a `StringBuilder` or `DefaultInterpolatedStringHandler` is clearer and the allocation savings rarely justify the bug surface.

char is not a character: Rune and Unicode scalars

The deepest correctness trap in .NET text is that `char` is a UTF-16 *code unit*, not a Unicode character. Any scalar above U+FFFF (emoji, many CJK extension characters, mathematical alphanumeric) is stored as a **surrogate pair** — two `char`s. So `"1".Length == 2`, and indexing `[0]` gives you half a character. Iterating with `foreach (char c in s)` or `s[i]` silently corrupts such text.

`System.Text.Rune` (.NET Core 3.0+, fully fleshed out by .NET 6) is the type that represents a single Unicode *scalar value*. Iterate with `EnumerateRunes()` to process real characters:

```

using System.Text;

static int CountScalars(string s)
{
    int count = 0;
    foreach (Rune r in s.EnumerateRunes())
        count++;
    return count;
}

static string ToUpperScalarwise(ReadOnlySpan<char> s)
{
    var sb = new StringBuilder(s.Length);
    foreach (Rune r in s.EnumerateRunes()) // span
        // overload also exists
        sb.Append(Rune.ToUpperInvariant(r).ToString());
    return sb.ToString();
}

// "a1b".Length == 4 (1 is a surrogate
// pair)
// CountScalars("a1b") == 3 (correct character
// count)

```

`Rune` exposes `Value` (the 21-bit scalar), `Utf16SequenceLength` (1 or 2), `Utf8SequenceLength`, and static classification helpers (`Rune.IsLetter`, `Rune.IsWhiteSpace`, `Rune.GetUnicodeCategory`). Use `Rune` whenever you reason about *characters* — truncating a display string, validating an identifier, counting “letters.”

Grapheme clusters: even `Rune` isn’t enough for “what the user sees”

A *user-perceived character* — a “grapheme cluster” — can be several scalars: `é` as `e` + combining acute accent, a flag emoji as two regional indicators, a family emoji joined by zero-width joiners. To split a string the way a human (or a backspace key) would, use `System.Globalization.StringInfo` and `GetTextElementEnumerator` / `StringInfo.GetNextTextElementLength` :

```
using System.Globalization;

static int CountGraphemes(string s)
{
    var e = StringInfo.GetTextElementEnumerator(s);
    int count = 0;
    while (e.MoveNext()) count++;
    return count;
}

// "👨".Length == 8 (UTF-16 code units)
// CountScalars(...) == 5 (scalars: 3 emoji
// + 2 ZWJ)
// CountGraphemes(...) == 1 (one perceived
// character)
```

The hierarchy to keep straight: **byte (UTF-8) c char (UTF-16 code unit) c Rune (scalar) c grapheme cluster (text element)**. Pick the level that matches your question. Buffer sizing? bytes or chars. “Is this a valid letter?” Rune. “Truncate to 10 characters for the UI” or “how many backspaces to delete this?” Grapheme clusters via `StringInfo`.

StringComparison: ordinal vs culture, and why it matters

The most damaging bug in this entire chapter is the silent one: comparing strings with the wrong semantics. `string.Equals`, `==`, `StartsWith`, `IndexOf`, `ToUpper`, and sorting each have a *default* that may be culture-sensitive, and culture-sensitive comparison is **non-ordinal, locale-dependent, and slow**. The canonical disaster is the “Turkish-i” problem: in the Turkish (`tr-TR`) culture, `"FILE".ToLower()` is `"file"` (dotless i), so a culture-aware case-insensitive compare of `"file"` and `"FILE"` *fails* on a Turkish machine.

The rule, stated plainly: **for program identifiers, protocol tokens, file paths, keys, and anything not shown to a human as sorted text, always pass**

`StringComparison.Ordinal` or

`StringComparison.OrdinalIgnoreCase` **explicitly**. Reserve

`CurrentCulture / InvariantCulture` for *linguistic* operations

— sorting a list for display, case-folding human names for search.

```
// WRONG: culture-sensitive default; breaks on tr-TR,
// and is slower.
if (header.StartsWith("Bearer ")) { /* ... */
}
if (ext.Equals(".PNG",
    StringComparison.CurrentCultureIgnoreCase)) {
    /* ... */ }

// RIGHT: ordinal – fast, deterministic, locale-
// independent.
if (header.StartsWith("Bearer ",
    StringComparison.Ordinal)) { /* ... */ }
if (ext.Equals(".png",
    StringComparison.OrdinalIgnoreCase)) { /* ...
    */ }

// Dictionary keyed by case-insensitive *identifier* –
// ordinal, not culture:
var byName = new Dictionary<string, int>
    (StringComparer.OrdinalIgnoreCase);
```

`OrdinalIgnoreCase` folds only the ASCII/Unicode simple case mapping, with no culture rules — it is both correct and roughly an order of magnitude faster than `CurrentCultureIgnoreCase` for typical keys. The analyzer [CA1307](#) / [CA1310](#) flags missing `StringComparison` arguments; treat them as errors in library code.

Span APIs via `MemoryExtensions`

`MemoryExtensions` provides span-based `StartsWith`, `IndexOf`, `Contains`, `Equals`, `Trim`, `Split`, and `SequenceEqual` that operate on `ReadOnlySpan<char>` with no substring allocation. Parsing and tokenizing over spans avoids the storm of small strings that `string.Split` produces. **C# 14 (.NET 10)** removes the friction here with first-class span conversions: a `string` now *implicitly* converts to `ReadOnlySpan<char>`, so you can pass a string straight to a span-taking method without `.AsSpan()`:


```

// .NET 9 introduced the ReadOnlySpan<char> Split
// overloads (Range destinations).
static (string scheme, string rest) SplitScheme(string
    url)
{
    ReadOnlySpan<char> s = url; // C#
    // 14: implicit string -> ReadOnlySpan<char>
    int i = s.IndexOf(":/",
        StringComparison.Ordinal);
    if (i < 0) return ("", url);
    return (s[..i].ToString(), s[(i +
        3)..].ToString()); // only two strings, no
        // temporaries
}

// Tokenize without allocating per-token strings (.NET
// 9 Split-into-Range destinations):
static int CountCsvFields(ReadOnlySpan<char> line)
{
    Span<Range> ranges = stackalloc Range[64];
    int n = line.Split(ranges, ','); //
        // fills ranges, returns count
    return n;
}

```

Be deliberate about ordinal comparisons here too: the `span.IndexOf(ReadOnlySpan<char>)` overload is *ordinal* by default (no culture), which is usually what you want for parsing — but pass the `StringComparison` overload when you mean it, to make intent explicit and to silence analyzers.

GeneratedRegex: source-generated regular expressions

Runtime `new Regex(pattern)` compiles the pattern via reflection-emitted IL (or a slower interpreter) on first use, costs start-up time, and is invisible to trimming/AOT. **.NET 7** introduced the `[GeneratedRegex]` source generator: you declare a `partial` method and the generator emits a fully-specialized, allocation-lean matcher *at compile time*. It is faster at steady state, has near-zero startup cost, is fully AOT/trim-compatible, and you can step through the generated code.

```

using System.Text.RegularExpressions;

public partial class SlugValidator
{
    // The generator implements this partial method at
    // compile time.
    [GeneratedRegex(@"^[a-z0-9]+(?:-[a-z0-9]+)*$",
        RegexOptions.CultureInvariant)]
    private static partial Regex SlugPattern();

    public static bool IsValidSlug(string candidate)
        =>
        SlugPattern().IsMatch(candidate);
}

```

Prefer `[GeneratedRegex]` for every *static, compile-time-known* pattern — it is the modern default. Use runtime `Regex` only when the pattern is genuinely dynamic (built from user input or config at runtime), in which case consider `RegexOptions.Compiled` *only* if the regex runs in a hot loop, since compilation has its own startup cost. Always pass `RegexOptions.CultureInvariant` unless you specifically need culture-aware character classes — another instance of the ordinal-by-default discipline. And remember regex matches over `char`, so a pattern like `.` matches one UTF-16 code unit; a `.` will not match an astral-plane emoji as a single character.

ArrayPool and stackalloc for transient text

When you build text in a tight loop and the result is short-lived, the buffer itself can dominate allocations. Two tools eliminate it. For small, bounded sizes, `stackalloc` puts a `Span<char>` on the stack — no GC involvement at all:

```

static bool TryFormatTimestamp(DateTime utc,
    Span<char> dest, out int written)
{
    // ISpanFormattable straight into a caller-
    // provided buffer.
    return utc.TryFormat(dest, out written, "0",
        null);
}

static string IsoNow()
{
    Span<char> buf = stackalloc char[33]; // 33
    // is the max for any "0" value (DateTime is 28,
    // DateTimeOffset 33)
    return DateTime.UtcNow.TryFormat(buf, out int n,
        "0", null)
        ? new string(buf[..n])
        : throw new InvalidOperationException();
}

```

Keep `stackalloc` sizes small and *constant* — a large or attacker-influenced `stackalloc` risks a stack overflow that you cannot catch. The conventional ceiling is a few hundred bytes; above that, rent from the pool:

```

using System.Buffers;

static string DescribeMany(ReadOnlySpan<int> values)
{
    char[] rented =
        ArrayPool<char>.Shared.Rent(values.Length *
            12);

    try
    {
        Span<char> buf = rented;
        int pos = 0;
        foreach (int v in values)
        {
            v.TryFormat(buf[pos..], out int w);
            pos += w;
            if (pos < buf.Length) buf[pos++] = ' ';
        }
        return new string(buf[..pos].TrimEnd());
    }
    finally
    {
        ArrayPool<char>.Shared.Return(rented); //
        ALWAYS return in finally
    }
}

```

The `ArrayPool` contract has two non-negotiable rules: **always Return in a finally**, and **never touch the array after returning it** (another caller may now own it). Rented arrays may be *larger* than requested, so slice to your logical length. Don't clear unless the buffer held secrets (`Return(rented, clearArray: true)` for sensitive data).

Worked example: a zero-allocation structured log line builder

Let us pull the threads together into one realistic component: a request-log formatter for a high-throughput web server. It must (1) format a fixed-shape line cheaply, (2) reuse a parsed template chosen at runtime by config, (3) compare the method and status as ordinal tokens, and (4) avoid heap traffic on the formatting path. We combine `CompositeFormat` (runtime-chosen template), `string.Create` / `TryFormat` (no intermediates), ordinal comparison, and `u8` tokens.

```

using System Buffers;
using System Globalization;
using System Text;

public sealed class RequestLogFormatter
{
    // Template comes from configuration, so it cannot
    // be a compile-time interpolation.
    // Parse it ONCE; reuse for the life of the
    // process.
    private readonly CompositeFormat _template;

    public RequestLogFormatter(string template) =>
        _template = CompositeFormat.Parse(template);
    // e.g. "{0} {1} -> {2} in {3}ms"

    public string Format(string method, string path,
        int status, double elapsedMs)
    {
        // Ordinal classification of a hot token -
        // never culture-sensitive.
        bool isWrite =
            method.Equals("POST",
                StringComparison.Ordinal) ||
            method.Equals("PUT",
                StringComparison.Ordinal) ||
            method.Equals("PATCH",
                StringComparison.Ordinal) ||
            method.Equals("DELETE",
                StringComparison.Ordinal);

        // StringBuilder +
        // AppendFormat(CompositeFormat) reuses the
        // parsed template.
        // The strongly-typed CompositeFormat
        // overloads only go up to three args
        // (TArg0..TArg2). With four holes, bind the
        // ReadOnlySpan<object?> overload -
        // a collection expression targets it directly
        // (the int boxes into object).
        var sb = new StringBuilder(64);
        ReadOnlySpan<object?> args =
            [method, path, status,
                elapsedMs.ToString("F1",
                    CultureInfo.InvariantCulture)];
        sb.AppendFormat(CultureInfo.InvariantCulture,
            _template, args);
        if (isWrite) sb.Append(" [write]");
        return sb.ToString();
    }
}

```

```

// The fixed-width correlation id never allocates
// an intermediate string.
public static string
    CorrelationId(ReadOnlySpan<char> prefix, long
seq) =>
    string.Create(prefix.Length + 16,
        (PrefixHolder: prefix.ToString(), seq),
        static (span, state) =>
        {
            state.PrefixHolder.CopyTo(span);
            // string.CopyTo(Span<char>) – no allocation
            Span<char> hex =
            span[state.PrefixHolder.Length..];
            state.seq.TryFormat(hex, out int w,
"x16"); // lower-hex, padded
            if (w < hex.Length)
                hex[w..].Fill('0');
        });
}

```

A couple of design points. We deliberately used `StringBuilder.AppendFormat(CompositeFormat)` rather than interpolation because the template is *data*, not code — interpolation requires a literal. The correlation id uses `string.Create` because its length is exactly known. And every comparison that classifies the HTTP method passes `StringComparison.Ordinal` explicitly: a request log must behave identically in Frankfurt and Ankara. On the truly hottest paths you could push the writer down to `ArrayPool<char>` and a `Utf8.TryWrite / u8`-token sink writing straight to the socket buffer; the version above trades a `StringBuilder` allocation for clarity, which is the correct default until a profiler says otherwise.

Bonus: an extension member for ergonomic span text (C# 14)

C# 14 (.NET 10) introduced *extension members* via `extension` blocks — you can now add extension *properties* and *static* members, not just methods, grouping them under a shared receiver declaration. A small, honest use is exposing ordinal helpers so call sites read cleanly without forgetting the comparison flag:

```

public static class StringTextExtensions
{
    extension(string s)
    {
        // extension property: user-perceived length,
        // computed correctly.
        public int GraphemeLength
        {
            get
            {
                var e =
System.Globalization.StringInfo.GetTextElement
Enumerator(s);
                int n = 0;
                while (e.MoveNext()) n++;
                return n;
            }
        }

        // extension method that bakes in ordinal
        // semantics.
        public bool EqualsOrdinal(string other) =>
            string.Equals(s, other,
StringComparison.Ordinal);
    }
}

// usage
bool same = "Bearer".EqualsOrdinal("bearer");    //
            false, no culture involved
int shown = "👨 hi".GraphemeLength;              // 4:
            family-emoji + space + h + i

```

Extension members are syntactic ergonomics over static methods; they generate the same kind of static helpers the older `this`-parameter extension methods do, so there is no runtime cost. Use them to encode *correct defaults* (ordinal comparison, grapheme-aware length) at the call site, which is itself a small defense against the comparison bugs that opened this section.

Summary and decision guide

The throughline of this chapter is matching the abstraction level to the question. Reach for **raw string literals** (C# 11) for any text with quotes, backslashes, or structure, and remember the closing-fence column defines the left margin. Use **u8 literals**

(C# 11) only at the byte boundary — protocols, signatures, wire tokens — never as storable values. Trust the compiler’s **interpolation lowering** through

`DefaultInterpolatedStringHandler` (C# 10) for the common case, write a **custom** `[InterpolatedStringHandler]` only when you must skip expensive argument evaluation (logging), and cache a `CompositeFormat` (.NET 8) when the template is chosen at runtime. Use `string.Create` and `TryFormat` to build known-length strings without intermediates, and `ArrayPool` / `stackalloc` for transient buffers — always returning rented arrays in a `finally`. Reason about real characters with `Rune` (and `StringInfo` grapheme clusters) rather than `char`, because surrogate pairs make `char`-indexing a latent corruption bug. Prefer `[GeneratedRegex]` (.NET 7) for every static pattern. And above all, pass `StringComparison.Ordinal` explicitly for identifiers, keys, and tokens — the culture-sensitive default is the most expensive habit in .NET text, in both performance and the bug it eventually ships to a machine whose locale you never tested.

Part II — Authoring Modern Types

6 · Primary Constructors and Collection Expressions

C# 12 quietly reshaped how everyday types are authored: primary constructors moved out of `record`-only territory and onto every `class` and `struct`, and collection expressions replaced a decade of `new[] { ... }`, `new List<T> { ... }`, and `Array.Empty<T>()` ceremony with a single uniform `[...]` syntax. C# 13 then closed the allocation gap with `params` collections over spans. This chapter covers what these features actually compile to, where they save allocations, and the surprisingly sharp edges that follow from “parameters are private captured state, not public members.”

Primary constructors beyond records

A *primary constructor* is the parameter list you write directly after a type name. It was introduced for `record` types in C# 9, and **C# 12 (.NET 8)** extended it to every `class` and `struct`. The crucial distinction from records: on a plain `class` or `struct`, primary-constructor parameters do **not** synthesize public properties. They are in scope for the entire type body as ordinary values, and the compiler captures any that are actually used into a hidden private field.

```
public class OrderService(IRepo repo,
    ILogger<OrderService> log)
{
    public async Task<Guid> PlaceAsync(Order order,
        CancellationToken ct)
    {
        log.LogInformation("Placing order for
            {Customer}", order.CustomerId);
        var id = await repo.InsertAsync(order, ct);
        log.LogInformation("Order {Id} persisted",
            id);
        return id;
    }
}
```

This is the canonical use case: dependency injection. The parameters `repo` and `log` are visible in every member, so the boilerplate trio — private readonly fields, a constructor, and `this.x = x` assignments — collapses into the header. The compiler observes that both parameters are referenced from an instance method and synthesizes two hidden fields to hold them across the object's lifetime. Decompiled, `OrderService` is indistinguishable from the hand-written version with two `private readonly` fields, except for one detail covered below: capture fields are **not** emitted `readonly`.

Capture is demand-driven. A parameter referenced only inside the constructor body (not from any other member) is *not* promoted to a field; it lives only as long as the constructor runs, exactly like a normal constructor argument.

```
public class Cache(int capacity, IClock clock)
{
    private readonly Dictionary<string, (DateTime
        stamp, object value)> _store = new(capacity);
    private readonly IClock _clock = clock;

    // `capacity` is used ONLY above (in the
    // initializer) – no hidden field for it.
    // `clock` is read only by the _clock initializer;
    // methods read _clock, never
    // the parameter, so `clock` is NOT captured –
    // exactly one field, _clock.
    public bool TryGet(string key, out object? value)
    {
        if (_store.TryGetValue(key, out var e) &&
            e.stamp > _clock.UtcNow)
        {
            value = e.value;
            return true;
        }
        value = null;
        return false;
    }
}
```

Here `capacity` is consumed by the field initializer `new(capacity)` and never referenced again, so it stays a constructor-local. `clock` is copied into the explicit `_clock` field

and all members read `_clock`, so the parameter `clock` is also never captured — there is exactly one field, `_clock`. This is the idiom to reach for when you want a `readonly` guarantee, because primary-constructor capture fields are mutable.

Mutability of captured parameters

On a `class` or a non-`readonly struct`, primary-constructor parameters are *assignable* inside the type, and the synthesized capture field is therefore not emitted `readonly`:

```
public class Counter(int start)
{
    public void Reset() => start = 0;    // legal: the
                                     parameter is mutable
    public int Current => start;
}
```

Whether this is a feature or a footgun depends on taste, but the consequence is concrete: if you need an immutable dependency (the overwhelmingly common DI case), do not rely on the capture field. Either copy into an explicit `private readonly` field, as `Cache` does above, or accept that another method could legally reassign the parameter. Analyzers will flag a captured-but-reassigned parameter, but the language permits it.

The one exception is a `readonly struct`: there the spec mandates that capture fields are emitted `readonly`, and access to a captured parameter carries the same restrictions as access to a `readonly` field, so `Reset() => start = 0;` would not compile inside a `readonly struct`. For classes and mutable structs, the mutability above is real.

A related compiler diagnostic is easy to trip over. If you both *initialize a member* from a parameter and *also capture* the same parameter in a member body, the compiler warns about double storage, because mutating the member would then silently diverge from what the captured reads see:

```
public class Person(string name)
{
    public string Name { get; set; } = name; //
        initialization
    public override string ToString() => name; //
        capture → warning: stored twice
}
```

Here `Name` and the captured `name` are two independent slots; reassigning `Name` does not change what `ToString()` returns. The fix is to pick one storage location — read `Name` from `ToString()` and drop the capture.

Combining with explicit constructors and `this(...)` chaining

A type with a primary constructor may still declare additional constructors, but every one of them **must** chain to the primary constructor via `: this(...)`. This guarantees the primary constructor (and any captured-field initialization) always runs.

```
public readonly struct Temperature
{
    public double Kelvin { get; }

    // Primary constructor: the single source of truth
    // for initialization.
    public Temperature(double kelvin)
    {
        if (double.IsNaN(kelvin) || kelvin < 0)
            throw new
                ArgumentOutOfRangeException(nameof(kelvin),
                    "Kelvin must be ≥ 0.");
        Kelvin = kelvin;
    }
}
```

That is the explicit-constructor form. To show the primary-constructor-plus-validating-secondary pattern on a `struct`, put the primary constructor in the header and chain the convenience constructor to it:

```

public readonly struct Celsius(double degrees)
{
    public double Degrees { get; } = degrees;

    // Validating secondary constructor chains to the
    // primary via : this(...).
    // It cannot touch `this` before the chain call,
    // so validation happens
    // on the argument, then the primary ctor runs the
    // field initializer.
    public Celsius(double degrees, bool validate)
        : this(validate && degrees < -273.15 ? throw
            new
            ArgumentOutOfRangeException(nameof(degrees),
                "Below absolute zero.")
            : degrees)
    {
    }
}

```

The constraint that secondary constructors must chain to the primary one — rather than to a `base(...)` or to nothing — is what makes primary constructors safe: there is exactly one initialization funnel. Note also the order: a property/field initializer that uses a primary-constructor parameter (`public double Degrees { get; } = degrees;`) runs as part of the primary constructor, *before* the body of any chaining constructor.

When NOT to use a primary constructor

- **You need `readonly` capture.** The capture field is mutable; reach for an explicit field.
- **You need validation on the parameter itself before any field is set.** A primary constructor has no body, so you cannot guard a header parameter inline except via the `throw` expression trick shown above, which gets unreadable fast. Prefer an explicit constructor when validation is non-trivial.
- **The type is a `record`.** There, the primary constructor already means “public init properties + deconstruct + equality.” Mixing record semantics with the class semantics described here confuses readers; keep the two mental models separate.

Pitfall: parameters are not public members

The single most common misconception: that `class Point(int x, int y)` exposes `x` and `y` the way a `record` would. It does not. On a plain class, `x` and `y` are private captured state with no accessor, no `init`, no deconstruction.

```
public class Point(int x, int y)
{
    public int X => x;    // you must surface them
                        // yourself
    public int Y => y;
}

var p = new Point(3, 4);
// var bad = p.x;      // ERROR CS1061: 'Point'
//                    // has no member 'x' (the capture
//                    // field, if one exists,
//                    // has an unspeakable mangled name)
Console.WriteLine(p.X); // 3 – only because you
                        // wrote the property
```

If you want public properties, write them. If you want a value type with structural equality and a deconstructor, use a `record class` or `record struct` — the primary constructor on a non-record type is deliberately a *lower-level* tool.

Pitfall: lambdas over captured parameters and the hidden `this` closure

This is the subtle one — and it is worth being precise, because the common folklore (“the value gets stored twice”) is wrong. When a primary-constructor parameter is captured into the type’s state, there is exactly **one** capture field. A lambda that reads the same parameter does not get its own private copy; the language spec requires the lambda and the type’s state to “refer to the same location for the parameter,” so the lambda closes over `this` and reads the shared instance field. The real costs are therefore (a) the closure (display-class) allocation for the lambda itself, and (b) a non-obvious consequence when that lambda lives in a *field/property initializer*: the compiler must capture

`this` into the closure **before the base constructor runs**, which it does, producing valid-but-unverifiable IL.

```
public class RetryPolicy(int maxAttempts, TimeSpan
    delay)
{
    // `delay` is read by THIS lambda, which is built
    // during the primary constructor.
    // Because the lambda lives in a field
    // initializer, capturing it forces `this`
    // into the closure before base-ctor execution.
    private readonly Func<int, TimeSpan> _backoff =
        attempt => TimeSpan.FromTicks(delay.Ticks *
            attempt);

    // `maxAttempts` is captured as an instance field
    // because RunAsync() reads it.
    public async Task RunAsync(Func<Task> action)
    {
        for (var i = 1; i <= maxAttempts; i++)
        {
            try { await action(); return; }
            catch when (i < maxAttempts) { await
                Task.Delay(_backoff(i)); }
        }
    }
}
```

Here `delay` is read only inside the lambda, so it is captured by that closure (via `this`); `maxAttempts` is read by `RunAsync`, so it becomes an ordinary instance capture field. There is no value duplication — but every parameter that crosses into a lambda still costs a heap closure allocation at construction time, on top of the object itself, and a lambda in a field initializer drags `this` into that closure early. The fix is to be deliberate: assign the parameter once to an explicit field and have everything — methods and lambdas — read that field, so the closure (if any) captures `this` and a plain field rather than a primary-constructor parameter:


```
public class RetryPolicy2(int maxAttempts, TimeSpan
    delay)
{
    private readonly int _maxAttempts = maxAttempts;
    private readonly TimeSpan _delay = delay;
    private Func<int, TimeSpan> Backoff => a =>
        TimeSpan.FromTicks(_delay.Ticks * a);
    // single field per value; the lambda reads
    // _delay, not the parameter,
    // and lives in a property getter, so no closure
    // is built in the ctor.
}
```

Moving the lambda from a field initializer to a property getter (`=> a => ...`) is the other half of the fix: a lambda in an *initializer* is constructed during the primary constructor and forces the early `this` capture described above, whereas a lambda in a getter is built lazily on first access. Inspect the IL (or just the allocation profile) when a hot-path type combines primary-constructor capture with lambdas — the closure allocation and any early `this` capture are easy to miss in source.

Collection expressions

C# 12 (.NET 8) introduced collection expressions: a single bracketed syntax, `[a, b, c]`, that target-types to arrays, `List<T>`, `Span<T>`, `ReadOnlySpan<T>`, and any type opted in via `CollectionBuilder`. The compiler chooses the cheapest construction strategy that satisfies the target type — there is no single fixed lowering.

```
int[]          a = [1, 2, 3];
List<int>      b = [1, 2, 3];
Span<int>      c = [1, 2, 3];
ReadOnlySpan<int> d = [1, 2, 3];
IEnumerable<int> e = [1, 2, 3];
HashSet<int>   f = [1, 2, 3];
```

Each line lowers differently. `a` is a plain array. `b` constructs a `List<int>` with the right capacity and adds the elements. `c` and `d` are the interesting cases: when the target is `Span<T>` / `ReadOnlySpan<T>` and the size is known and small,

the compiler can place the backing storage **on the stack** (effectively a `stackalloc` /inline-array buffer) with zero heap allocation — see the span section below. `e` (an interface) gets an efficient compiler-internal array-backed implementation. `f` constructs a `HashSet<int>`.

Spread elements

The spread element `..other` inlines the contents of another collection into the expression. Spread works with any enumerable source, and you can freely mix spreads with singleton elements.

```
int[] head = [1, 2];
int[] tail = [5, 6];
int mid = 3;

int[] all = [..head, mid, 4, ..tail]; // [1, 2, 3,
                                     4, 5, 6]
```

The compiler computes the final length where it can (here both spread operands are arrays with a known `Length`), allocates one destination array of exactly six elements, and copies each segment — `head` via `Array.Copy` / `Span.CopyTo`, the singletons by index, then `tail`. There is no intermediate `List<int>`, no repeated reallocation. When a spread source's length is *not* known up front (e.g. spreading a lazy `IEnumerable<T>`), the compiler falls back to a growable builder, so a spread of an unbounded sequence is no cheaper than the equivalent `List<T>` accumulation — keep that in mind on hot paths.

```
// Length known at compile time for both operands →
// single exact-size array, two CopyTo calls.
ReadOnlySpan<int> left = [10, 20];
ReadOnlySpan<int> right = [30, 40, 50];
int[] merged = [..left, ..right]; // one
// int[5], zero intermediate collections
```

Building a `ReadOnlySpan<int>` with no heap allocation

```
ReadOnlySpan<int> data = [1, 2, 3];
int sum = 0;
foreach (var x in data) sum += x;           // sum == 6,
                                           no array on the heap
```

For a `ReadOnlySpan<T>` (or `Span<T>`) target with a compile-time-known, small element count of blittable values, the compiler allocates the backing buffer on the stack via an inline array and points the span at it — equivalent to `stackalloc int[3]` followed by element assignment. This is the headline performance win of collection expressions: `ReadOnlySpan<int> data = [1, 2, 3];` is allocation-free, whereas the old `int[] data = {1, 2, 3};` always heap-allocated. The usual `stackalloc` caveat applies: the buffer's lifetime is the enclosing method, so never return such a span or let it escape, and keep the element count bounded so you do not blow the stack.

In **C# 14 (.NET 10)**, first-class implicit span conversions make this even smoother. An `int[]` now implicitly converts to `ReadOnlySpan<int>` (and arrays convert to `Span<T>`; `string` converts to `ReadOnlySpan<char>`), so passing array-typed collection-expression results to span-typed parameters no longer needs an explicit `.AsSpan()`:

```
static int Sum(ReadOnlySpan<int> values)
{
    int total = 0;
    foreach (var v in values) total += v;
    return total;
}

int[] xs = [1, 2, 3, 4];
int s = Sum(xs); // C# 14: implicit int[] →
                 ReadOnlySpan<int>, no .AsSpan() needed
```

Target typing in detail

A collection expression has no type of its own; it acquires one from context. The rules in priority order: if the target is an array,

`Span<T>`, `ReadOnlySpan<T>`, or a type bearing `[CollectionBuilder]`, the compiler uses that. If the target is `IEnumerable<T>`, `IReadOnlyList<T>`, `IReadOnlyCollection<T>`, `IList<T>`, or `ICollection<T>`, it synthesizes a suitable concrete type. If the target is a concrete collection with an applicable constructor and `Add / AddRange` surface (or an `init` indexer), it uses that. Because there is no standalone type, `var x = [1, 2, 3];` does **not** compile — there is nothing to infer from. You must state the target:

```
// var oops = [1, 2, 3];           // ERROR CS9176: no
//                               // target type for the collection expression
List<int> ok = [1, 2, 3];          // fine
ReadOnlySpan<char> chars = ['a', 'b'];
```

Empty collection expressions: `[]` vs `Array.Empty<T>()`

The empty form `[]` target-types like any other collection expression, and for empty arrays Roslyn emits `Array.Empty<T>()` rather than allocate a fresh zero-length array — so `[]` is the modern, allocation-free way to produce an empty array, identical in IL to writing `Array.Empty<int>()` yourself. (The same `Array.Empty<T>()` lowering applies when the empty target is `IEnumerable<T>`, `IReadOnlyCollection<T>`, or `IReadOnlyList<T>`, since a cached empty array satisfies all three.)

```
int[] none = [];                // emits
//                               Array.Empty<int>() - cached singleton, no
//                               allocation
List<int> emptyList = [];        // new List<int>()
//                               (lists are not cached; this allocates)
ReadOnlySpan<int> emptySpan = []; // default(ReadOnlySpan<int>) - empty, no
//                               allocation
```

Prefer `[]` over `Array.Empty<T>()` for new code: it is shorter, target-types automatically, and produces the same cached singleton for arrays. Do not assume `[]` is free for *every* target — `List<int> x = [];` still allocates an empty list, and `new()`

would be equally cheap. The “free” guarantee is specifically for the array and span cases.

Making your own type collection-expression-constructible

To let a custom type be built from `[...]`, mark it with the `[CollectionBuilder]` attribute (C# 12), naming a static factory method that takes a single `ReadOnlySpan<T>` and returns the collection. The builder method must be static, accessible at the call site, have arity matching the collection type, and return a type with an identity/implicit-reference/boxing conversion to the collection type.

```

using System.Runtime.CompilerServices;

[CollectionBuilder(typeof(RingBuffer),
    nameof(RingBuffer.Create))]
public sealed class RingBuffer<T> :
    IReadOnlyCollection<T>
{
    private readonly T[] _items;
    private int _head;
    public int Count { get; private set; }

    internal RingBuffer(int capacity) => _items = new
        T[capacity];

    public void Add(T item)
    {
        _items[_head] = item;
        _head = (_head + 1) % _items.Length;
        if (Count < _items.Length) Count++;
    }

    public IEnumerator<T> GetEnumerator()
    {
        // enumerate oldest → newest
        int start = (_head - Count + _items.Length) %
            _items.Length;
        for (int i = 0; i < Count; i++)
            yield return _items[(start + i) %
                _items.Length];
    }

    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
        => GetEnumerator();
}

// The builder type. The factory's single parameter
// MUST be ReadOnlySpan<T>.
public static class RingBuffer
{
    public static RingBuffer<T> Create<T>
        (ReadOnlySpan<T> items)
    {
        var rb = new RingBuffer<T>
            (Math.Max(items.Length, 1));
        foreach (var item in items)
            rb.Add(item);
        return rb;
    }
}

```

With that in place, the collection-expression syntax flows straight into `RingBuffer.Create` :

```
RingBuffer<int> rb = [1, 2, 3, 4];           // →  
                                RingBuffer.Create(ReadOnlySpan<int>[1,2,3,4])  
RingBuffer<int> merged =[..rb, 5, 6];       // spread  
                                the buffer, then append two more  
foreach (var x in merged) Console.Write($"{x} "); //  
1 2 3 4 5 6
```

The compiler materializes the bracketed elements into a `ReadOnlySpan<T>` (stack-allocated when small and blittable, otherwise a temporary array) and hands it to `Create`. Because the factory receives a span, *you* control how the final collection is built — no requirement that it be a `List<T>` or even mutable. This is how `ImmutableArray<T>` and friends in the BCL accept collection expressions. Note the arity rule: the generic `Create<T>` matches the generic `RingBuffer<T>`; a non-generic builder for a generic collection would not bind.

`params` collections

C# 13 (.NET 9) generalized `params` beyond arrays. A `params` parameter may now be `ReadOnlySpan<T>`, `Span<T>`, `IEnumerable<T>`, or any collection type the compiler knows how to build (the same set collection expressions target). The headline win is `params ReadOnlySpan<T>`: the call site can pass arguments inline with **no array allocation**, because the compiler stack-allocates the span backing.

```

public static class Log
{
    // C# 13: params ReadOnlySpan<T> – no per-call
    // array on the heap.
    public static void Write(string template, params
        ReadOnlySpan<object?> args)
    {
        Console.Write(template);
        Console.Write(" :: ");
        for (int i = 0; i < args.Length; i++)
            Console.Write($"[{i}]={args[i]} ");
        Console.WriteLine();
    }
}

Log.Write("user login", "alice", 42, true);
// Pre-C#13 (params object[]) this allocated a 3-
// element object[] on every call.
// With params ReadOnlySpan<object?> the backing
// buffer lives on the stack – the
// object[] is gone. Note: with element type object?,
// boxing 42 and true still
// allocates two boxes; the span eliminates the ARRAY,
// not the boxing. Use a
// concrete element type (e.g. params
// ReadOnlySpan<int>) to be truly box-free.

```

Compare the three `params` shapes:

```

static int SumArray(params int[] xs) { int
    s = 0; foreach (var x in xs) s += x; return s;
}
static int SumSpan (params ReadOnlySpan<int> xs){ int
    s = 0; foreach (var x in xs) s += x; return s;
}
static int SumEnum (params IEnumerable<int> xs) { int
    s = 0; foreach (var x in xs) s += x; return s;
}

SumArray(1, 2, 3); // heap int[3] per call
SumSpan (1, 2, 3); // stack-allocated buffer, zero
// heap allocation ← prefer for hot paths
SumEnum (1, 2, 3); // compiler builds a small
// enumerable; flexible but not allocation-free

```

For a frequently called variadic API, `params ReadOnlySpan<T>` is the right default in C# 13+. `params IEnumerable<T>` is for APIs that want to also accept an already-materialized sequence

without a wrapping array — but it loses the stack-allocation benefit, so do not reach for it on a hot path.

Pitfall: overload resolution between `params Span<T>` and `params T[]`

When you *add* a `params ReadOnlySpan<T>` overload alongside a legacy `params T[]` one — a common move when modernizing a library — overload resolution can surprise you, and the rules shifted in **C# 14**.

```
static void Print(params int[] xs)           =>
    Console.WriteLine($"array {xs.Length}");
static void Print(params ReadOnlySpan<int> xs) =>
    Console.WriteLine($"span {xs.Length}");

Print(1, 2, 3);    // expanded form: the span
                  // overload is preferred (no allocation)
Print([1, 2, 3]);  // collection expression: also
                  // prefers the span overload

int[] existing = [1, 2, 3];
Print(existing);   // passes an EXISTING array →
                  // array overload (it's an exact match)
                  // ...but in C# 14, implicit
                  // int[]→ReadOnlySpan<int> makes the span
                  // overload applicable too; the
                  // array overload still wins as the
                  // better match for an array
                  // argument. Confirm with your analyzer.
```

The friction is real: C# 14's first-class implicit span conversions make span-parameter overloads applicable in many places they previously were not, so a call that used to bind unambiguously to a `T[]` overload (or to a LINQ `Enumerable` method) can now have a *new* span candidate in the mix. Microsoft documents this as a deliberate **breaking change** to overload resolution; for example, inside an `Expression<...>` lambda, `array.Contains(x)` bound to `Enumerable.Contains` in C# 13 but binds to `MemoryExtensions.Contains` (which takes a `ReadOnlySpan<T>`) in C# 14 — and that throws at runtime if you `Expression.Compile(preferInterpretation: true)`. The defenses:

- When you own both overloads, give the one you want chosen higher priority with `[OverloadResolutionPriority]` (introduced in **C# 13**) rather than relying on tie-break intuition.
- At a problematic call site, cast the argument to the exact parameter type, or call the method as an explicit static invocation, to force the intended binding.
- Audit `Expression`-tree code after upgrading to C# 14. The C# compiler will happily *build* an expression tree whose node references a span-based BCL overload — it does not error at compile time. The failure is deferred: such a tree throws at **runtime** when you interpret it via `Expression.Compile(preferInterpretation: true)` (expression-tree interpretation cannot deal with `ref struct` /span parameters). Microsoft’s recommended fixes are to force the non-span binding —

```
((IEnumerable<int>)array).Contains(num) ,
array.AsEnumerable().Contains(num) ,
Enumerable.Contains(array, num) .
```

Collection expressions, inline arrays, and

`stackalloc`

The mechanism that makes `ReadOnlySpan<int> data = [1, 2, 3]` allocation-free is the inline array — the `[InlineArray]` struct feature also introduced in **C# 12**. An inline array is a fixed-size value type whose elements are laid out contiguously and addressable as a span. When the compiler lowers a small span-typed collection expression, it conjures exactly such a buffer on the stack and exposes it as the span. You normally never see the inline array; it is an implementation detail. But you *can* author one, and collection expressions interoperate with it:

```

using System.Runtime.CompilerServices;

[InlineArray(4)]
public struct Buffer4<T>
{
    private T _element0;    // the single field;
                           // [InlineArray(4)] makes room for 4
}

// A span-typed collection expression lowers through a
// *compiler-synthesized*
// inline-array buffer on the stack – you never name
// it. (You cannot initialize
// your own Buffer4<T> with [...] in C# 12; that
// remains an open proposal.)
// This line is allocation-free, and the Span is
// mutable:
Span<int> scratch = [0, 0, 0, 0];
scratch[1] = 42;      // scratch is now [0, 42, 0,
                      // 0]

```

The relationship to `stackalloc` is that a span-typed collection expression is, in effect, a friendlier `stackalloc` with element initializers and bounds known to the compiler. Reach for an explicit `stackalloc` only when you need a runtime-computed length; for a fixed small set of values, `Span<int> s = [a, b, c];` is clearer and equally cheap. As always with stack buffers: bound the size (a few hundred elements at most), never let the span escape the method, and fall back to a pooled array (`ArrayPool<T>`) for large or variable sizes.

Worked example: tying it together

A small request-pipeline router that uses a primary constructor for its dependencies, builds its middleware list with a spread collection expression, and exposes a hot-path `Dispatch` that takes `params ReadOnlySpan<...>` to avoid per-call allocation.

```

using System.Diagnostics;

public interface IMiddleware
{
    ValueTask<int> InvokeAsync(string path,
        Func<ValueTask<int>> next);
}

// Primary constructor captures the two dependencies
// as private state (C# 12).
public sealed class Router(IMiddleware[] global,
    ILogger log)
{
    // Spread composition: base middleware + per-route
    // extras, single exact-size array (C# 12).
    public IMiddleware[] BuildChain(params
        ReadOnlySpan<IMiddleware> routeSpecific)
        => [..global, ..routeSpecific];    //
        // one allocation, no intermediate List

    public async ValueTask<int> Dispatch(string path,
        params ReadOnlySpan<IMiddleware> extra)
    {
        var chain = BuildChain(extra);
        var sw = Stopwatch.GetTimestamp();

        Func<ValueTask<int>> terminal = () =>
            ValueTask.FromResult(200);
        // fold the chain right-to-left into a single
        // delegate
        for (int i = chain.Length - 1; i >= 0; i--)
        {
            var mw = chain[i];
            var next = terminal;
            terminal = () => mw.InvokeAsync(path,
                next);
        }

        int status = await terminal();
        log.Info($"{path} → {status} in
            {Stopwatch.GetElapsedTime(sw).TotalMilliseconds
            s:F2} ms");
        return status;
    }
}

public interface ILogger { void Info(string message); }
public sealed class ConsoleLogger : ILogger { public
    void Info(string m) => Console.WriteLine(m); }

// usage

```

```

var router = new Router(
    global: [new AuthMiddleware(), new
              MetricsMiddleware()], // collection expr →
              IMiddleware[]
    log:      new ConsoleLogger());

// params ReadOnlySpan<IMiddleware>: these two
//        middlewares are passed with no heap array.
await router.Dispatch("/orders", new
    RateLimitMiddleware(), new CacheMiddleware());

sealed class AuthMiddleware      : IMiddleware {
    public ValueTask<int> InvokeAsync(string p,
        Func<ValueTask<int>> n) => n(); }
sealed class MetricsMiddleware   : IMiddleware {
    public ValueTask<int> InvokeAsync(string p,
        Func<ValueTask<int>> n) => n(); }
sealed class RateLimitMiddleware : IMiddleware {
    public ValueTask<int> InvokeAsync(string p,
        Func<ValueTask<int>> n) => n(); }
sealed class CacheMiddleware     : IMiddleware {
    public ValueTask<int> InvokeAsync(string p,
        Func<ValueTask<int>> n) => n(); }

```

Three chapter ideas converge here. The primary constructor `Router(IMiddleware[] global, ILogger log)` captures `global` and `log` as private fields — note that, per the pitfall above, they are *not* public; nothing outside `Router` can read `router.global`. The chain is assembled with `[..global, ..routeSpecific]`, a single exact-size array built by two `CopyTo` calls rather than a `List<IMiddleware>` that grows and reallocates. And `Dispatch / BuildChain` accept `params ReadOnlySpan<IMiddleware>`, so the two extra middlewares at the call site are passed through a stack buffer with zero array allocation on the way in — the only heap allocation is the final composed `chain` array, which is genuinely needed because the folded delegates capture it.

One honest caveat in that example: the delegate-folding loop allocates a closure per middleware (each `() => mw.InvokeAsync(path, next)` captures `mw`, `path`, and `next`). The `params ReadOnlySpan<T>` and spread optimizations save the *argument* and *chain-array* allocations, not the closures inherent to building a middleware pipeline. Knowing precisely which allocations a feature removes — and which it leaves un-

touched — is the difference between cargo-culting collection expressions and using them where they pay.

Summary and guidance

- **Primary constructors (C# 12)** on classes/structs are private captured state, not public members. Use them for DI and concise initialization; copy into explicit `readonly` fields when you need immutability; route all secondary constructors through `: this(...)`.
- **Watch lambda capture:** a parameter read inside a lambda shares the *same* capture field as the rest of the type (no value duplication), but the lambda still costs a closure allocation — and a lambda in a field/property *initializer* drags `this` into that closure before the base constructor runs. Build such lambdas in a property getter, or read a plain explicit field, on hot paths.
- **Collection expressions (C# 12)** target-type to the cheapest construction for the destination. `ReadOnlySpan<T> x = [...]` is allocation-free via inline arrays; `int[] x = []` emits the cached `Array.Empty<T>()`; spreads of known-length sources build one exact-size buffer.
- **[CollectionBuilder] (C# 12)** opens your own types to `[...]` via a static `Create(ReadOnlySpan<T>)` factory — you decide how the final collection is built.
- **params collections (C# 13)** — prefer `params` `ReadOnlySpan<T>` on hot variadic APIs to eliminate the per-call array. When mixing with a legacy `params T[]` overload, pin the intended binding with `[OverloadResolutionPriority]`, and re-audit overload resolution and `Expression` trees after moving to **C# 14**, whose first-class implicit span conversions widen where span overloads apply.

Sources consulted for C# 14 / .NET 10 behavior: [Microsoft Learn — Collection expressions](#), [Microsoft Learn — First-class span types \(C# 14 proposal\)](#), [Breaking change: C# 14 overload resolu-](#)

tion with span parameters, and Andrew Lock — adding collection-expression support to your own types.

7 · The field Keyword and Modern Property Design

For two decades the C# auto-property has been an all-or-nothing bargain: accept a compiler-managed backing field and zero behavior, or hand-write the field and both accessors. C# 14 dissolves that dichotomy with the `field` contextual keyword, which lets a single accessor reach the synthesized backing field while the compiler still generates everything else. This chapter dissects the feature's mechanics, its surprisingly subtle interaction with nullable reference analysis, the upgrade hazard it introduces, and where it belongs in a decision matrix alongside auto-properties, full properties, and the source-generator patterns enabled by C# 13's partial properties.

The shape of the problem before C# 14

An auto-property is a contract with the compiler. When you write `public string Name { get; set; }`, Roslyn synthesizes a private field — historically named something like `<Name>k__BackingField`, an unspeakable name precisely so your code can never collide with it — and emits trivial accessors that read and write it. The moment you need *one line* of behavior in *one* accessor, the bargain collapses. You must declare the field yourself, name it, and write *both* accessors, even the trivial one:

```
public sealed class Customer
{
    private string _name = "";

    public string Name
    {
        get => _name;
        set => _name = value?.Trim()
            ?? throw new
                ArgumentException(nameof(value));
    }
}
```


The `get` here is pure boilerplate. It exists only because declaring an explicit field forces you to wire up both ends. Multiply this across a domain model with twenty validated properties and you have a hundred lines of mechanical field plumbing whose only purpose is to feed two-line setters. The explicit field also leaks: nothing stops a method elsewhere in the class from writing `_name = somethingUntrimmed`, silently bypassing the validation the property was supposed to guarantee.

The field contextual keyword (C# 14)

Introduced in C# 14 (.NET 10, November 2025), the `field` contextual keyword refers, inside a property accessor body, to the compiler-synthesized backing field for that property. (It is a keyword in `get`, `set`, and `init` accessors of *properties only* — not in indexer accessors, and not in event `add` / `remove` accessors, where the language design team deliberately left no synthesized backing field.) You write the accessor that needs behavior; the compiler supplies the field *and* any accessor you leave in auto form.

```
public sealed class Customer
{
    public string Name
    {
        get;
        set => field = value?.Trim()
            ?? throw new
                ArgumentException(nameof(value));
    }
}
```

The `get;` is a genuine auto-getter — the compiler emits `return <synthesized field>;`. The `set` reaches the *same* synthesized field through `field`. There is exactly one backing field, shared by both accessors, and it is as unspeakable as ever: no other member of `Customer` can name or mutate it, so the trimming invariant is airtight. This is the canonical motivating example from the language design, and it is worth internalizing as the feature's center of gravity: **auto-get with a validating set**.

`field` is *contextual* — it is a keyword only when used as a primary expression inside a property `get` / `set` / `init` accessor body. Everywhere else `field` remains an ordinary identifier. Inside such an accessor it binds to the backing field with the property's type. It is a real, addressable, `ref`-able storage location: you can take `ref field`, pass it to an `Interlocked` method (e.g. `Interlocked.CompareExchange(ref field, ..., ...)`), or use it as the target of a compound assignment. That `ref`-ability is precisely what makes the MVVM `SetProperty(ref field, value)` pattern later in this chapter possible.

A property may use `field` in one accessor, both, or neither:

```
public sealed class Thermostat
{
    // Both accessors use field: clamp on write,
    // expose raw on read.
    public double TargetCelsius
    {
        get => field;
        set => field = Math.Clamp(value, 5.0, 35.0);
    }

    // Only set uses field; get is auto. Functionally
    // identical for the getter.
    public int Retries
    {
        get;
        set => field = value < 0
            ? throw new
                ArgumentOutOfRangeException(nameof(value))
                : value;
    }
}
```

Pitfall: `get => field; set => field = value;` is just an **auto-property**

The single most common misreading of this feature is believing that *touching* `field` adds behavior. It does not.

```
// This is a verbose, fully-equivalent rewrite of
// `public int Count { get; set; }`.
// It adds NOTHING. The JIT inlines both accessors to
// direct field access.
public int Count
{
    get => field;
    set => field = value;
}
```

The compiler emits the same IL as a plain auto-property: two trivial accessors over one backing field, both inlined by the JIT to a single `ldfld` / `stfld`. If you find yourself writing accessors whose entire body is `field` / `field = value`, delete them and write `{ get; set; }`. `field` earns its place only when at least one accessor contains real logic. Using it ceremonially is pure noise.

Pitfall: `field` still allocates a backing field

`field` is not a calculated property. It does not make storage disappear. Every property that mentions `field` (or is a plain auto-property) carries one instance field's worth of memory per object — eight bytes for a reference, or `sizeof(T)` for a value type, plus alignment. If your property is genuinely *computed* from other state, do **not** introduce `field`:

```
public sealed class Rectangle(double width, double
    height)
{
    public double Width { get; } = width;
    public double Height { get; } = height;

    // Correct: no storage, recomputed each access. Do
    // NOT use `field` here.
    public double Area => Width * Height;
}
```

Reaching for `field` in `Area` would allocate a never-read field and produce a stale-able value. `field` is for properties that *have* state but need *behavior around* it. Computed properties have no state; expression-bodied members are still the right tool.

Lazy initialization and caching with field

A frequent reason to write a full property historically was lazy, cached computation. `field` collapses that pattern, and it pairs beautifully with C# 8's null-coalescing assignment (`??=`) for reference types:

```
public sealed class ReportBuilder(ReadOnlyList<Order>
    orders)
{
    // Computed once on first access, cached in the
    // synthesized field thereafter.
    public IReadOnlyDictionary<string, decimal>
        RevenueByRegion =>
        field ??= orders
            .GroupBy(o => o.Region)
            .ToDictionary(g => g.Key, g => g.Sum(o =>
                o.Total));
}
```

`field ??= ...` reads the backing field; if it is null, it evaluates the right side, stores it, and yields it. This is an *expression-bodied get-only property* whose backing field the compiler still synthesizes — there is no `set` at all, yet `field` is writable from within the getter. That is a deliberate and important capability: a get-only `field`-backed property can mutate its own backing store.

For value-typed caches the `field ??= ...` idiom does **not** apply, and this is a hard rule worth stating precisely: the synthesized backing field always has the **same type** as the property. The only thing that can differ between field and property is the *nullable annotation*, and that adjustment is made only for reference types (see the next section). A property of type `double` therefore has a backing field of type `double` — there is no implicit `double?` cache cell, and `field ??= data.Average()` will not even compile, because `??=` requires a left operand that can be `null`. To lazily cache a value type you must make the *property* expose a nullable, or carry an explicit `bool` “computed” flag:

```

public sealed class StatisticalSample(double[] data)
{
    private bool _meanComputed;

    // double has no null sentinel; gate the cache
    // with an explicit flag.
    public double Mean
    {
        get
        {
            if (!_meanComputed)
            {
                field = data.Average();
                _meanComputed = true;
            }
            return field; // field is
                          // of type double, same as the property
        }
    }
}

```

The backing field here is a plain `double`; `field` lets the getter cache into it without a hand-named field, but the *gate* still needs its own storage. This is exactly the case where `field`'s convenience is partial: it saves the value cell, not the flag.

A caution on threading: `field ??= expr` is **not** atomic and **not** thread-safe. Under concurrent first-access, the factory may run on multiple threads and different callers may observe different instances until one write wins. For shared mutable state across threads, this caching idiom is wrong; reach for `Lazy<T>` or `LazyInitializer.EnsureInitialized`, which provide publication guarantees `field ??=` does not:

```

using System.Collections.Frozen; // FrozenSet<T> is
                                not in the default implicit usings

public sealed class ConfigCache
{
    private readonly Lazy<FrozenSet<string>>
        _allowList =
        new(static () => LoadAllowList(),
            LazyThreadSafetyMode.ExecutionAndPublication);

    public FrozenSet<string> AllowList =>
        _allowList.Value;

    private static FrozenSet<string> LoadAllowList()
        => /* ... */
        FrozenSet<string>.Empty;
}

```

This is a case where the *explicit* field is correct precisely because you need a richer type (`Lazy<T>`) than the property exposes. `field` does not help when the storage type and the property type must diverge structurally.

field and nullable reference analysis

This is the subtlest corner of the feature and the one most likely to surprise experts. Consider:

```

#nullable enable
public sealed class Profile
{
    public string DisplayName => field ??=
        ComputeDefault();

    private static string ComputeDefault() =>
        "Anonymous";
}

```

The property type is non-nullable `string`. But the backing field starts as `null` (it is `default(string)`), and the getter assigns it lazily. If the compiler naively gave the field the property's annotation — non-nullable `string` — it would have to warn that the field is null before first assignment, which would defeat the

pattern. If it always made the field nullable, you would lose warnings everywhere the pattern *isn't* null-resilient.

C# 14 resolves this with **null-resilience analysis**, performed only when the property is a not-annotated (non-nullable) reference type. (The backing field's *type* is always identical to the property's; only its **nullable annotation** is what this analysis decides.) The compiler runs the getter through nullable flow analysis under two assumptions: one where the backing field is annotated (nullable), one where it is not-annotated (non-nullable). The rule, per the feature spec, is precise: if the *annotated* pass produces a nullable diagnostic that the *not-annotated* pass did not, the property is **not** null-resilient; otherwise it is. (The implementation short-circuits: it runs the annotated pass first, and if that yields no diagnostics at all the property is resilient and the second pass is skipped.) A *null-resilient* property gets an **annotated (nullable)** backing field; a non-resilient one keeps the property's own not-annotated annotation.

In the `Profile` example, the getter `field ??= ComputeDefault()` never dereferences a possibly-null `field` (the `??=` guarantees a non-null result), so it is null-resilient: the backing field's annotation becomes *annotated* — it behaves exactly like `string?` inside the accessor — no warning fires, and callers see a non-null `string`. Contrast:

```
#nullable enable
public sealed class Broken
{
    // Getter dereferences field directly; NOT null-
    // resilient.
    // Backing field keeps the non-nullable
    // annotation, so the compiler
    // warns that it may be null on the path before
    // first set.
    public string Tag
    {
        get => field.ToUpperInvariant(); // CS8602:
        // dereference of a possibly-null reference
        set => field = value;
    }
}
```

Because the getter dereferences `field` without a null guard, the property is not null-resilient; the backing field keeps the non-nullability annotation, the compiler issues **CS8602** for the dereference, and (if `Tag` is never initialized in a constructor)

CS9264, the field-backed-property variant of the “non-nullable member must be initialized” warning, which spells out the three fixes in its own message. The fix is to initialize in a constructor, provide a `field` initializer, make the getter resilient (`field ?? = ...`), or annotate the property `string?`. This null-resilience analysis is unique to `field`-backed properties (and ordinary auto-properties) and is the mechanism that makes the lazy-cache idiom warning-clean without annotations.

You can also annotate the property explicitly and the field follows:

```
public string? Nickname { get => field; set => field =  
    value?.Trim(); }  
// Property is annotated string?, so the field is  
    string?. No resilience analysis.
```

field initializers and required members

A `field`-backed property can carry an initializer, exactly like an auto-property, and the initializer targets the backing field:

```
public sealed class Session  
{  
    public DateTimeOffset Started { get; } =  
        DateTimeOffset.UtcNow;  
  
    public string Token  
    {  
        get;  
        set => field =  
            string.IsNullOrWhiteSpace(value)  
                ? throw new ArgumentException("Token  
                    required.", nameof(value))  
                : value;  
    } = Guid.NewGuid().ToString("N"); // initializer  
        feeds the backing field directly  
}
```


A critical subtlety: a property **initializer assigns the backing field directly and does NOT run the `set` accessor's validation**. In `Session`, the `= Guid.NewGuid()...` bypasses the whitespace check. That is usually fine (you control the literal), but never assume an initializer is validated — it is a raw field write.

`field` composes cleanly with `required` (C# 11). A `required field`-backed property forces callers to assign through the object initializer, which *does* invoke the `set` accessor and therefore *does* run validation:

```
public sealed class Invoice
{
    public required string Number
    {
        get;
        set => field = value?.Trim() is { Length: > 0 } v
            ? v
            : throw new ArgumentException("Invoice
            number required.");
    }
}

// var inv = new Invoice(); //
// CS9035: required member not set
var ok = new Invoice { Number = " INV-42 " }; //
// set runs: stored as "INV-42"
```

This is the cleanest way to express *validated immutable construction*: combine `required`, an `init`-or-`set` accessor that validates, and let `field` hold the result.

init accessors with field

The `init` accessor (C# 9) plus `field` is the idiomatic way to build immutable types whose values are validated *once*, at construction, and never again:

```

public sealed class GeoPoint
{
    public required double Latitude
    {
        get;
        init => field = value is >= -90 and <= 90
            ? value
            : throw new
                ArgumentOutOfRangeException(nameof(value));
    }

    public required double Longitude
    {
        get;
        init => field = value is >= -180 and <= 180
            ? value
            : throw new
                ArgumentOutOfRangeException(nameof(value));
    }
}

var p = new GeoPoint { Latitude = 50.08, Longitude =
    14.44 }; // Prague, validated
// p.Latitude = 0;
// CS8852: init-only

```

After construction the object is deeply immutable: there is no `set`, the backing field is unreachable except through the validated `init`, and the `required` modifier guarantees both coordinates were supplied. Before `field`, this required two explicit backing fields and four lines of getter boilerplate. The behavior is identical, the IL is identical, but the source is a third the size and the invariant is unforgeable.

Naming collisions and the `@field` escape

Because `field` was an ordinary identifier for the entire prior history of C#, making it contextual is a **breaking change** — the highest-stakes part of adopting C# 14 in an existing codebase. The language designers deliberately scoped the break as narrow as possible: `field` is a keyword *only when used as a primary expression* inside a property `get` / `set` / `init` accessor body. Elsewhere it stays an identifier. But inside such an accessor, the bare token `field` now binds to the synthesized backing field,

not to any field, local, parameter, or type named `field` you may have in scope.

```
public sealed class Plot
{
    private int field = 7;           // a real member
                                   // named 'field' (legal, legacy)

    public int First
    {
        // Pre-C# 14 this returned this.field (== 7).
        // Under C# 14 the bare
        // `field` is the SYNTHESIZED backing field of
        // First (default 0)!
        get => field;               // WARNING CS9258 in C# 14:
                                   // 'field' now refers to the backing field
    }
}
```

This is the dangerous case: the code still compiles, but `First` now returns `0` (the synthesized field's default) instead of `7`. The compiler raises warning **CS9258** wherever a bare `field` inside an accessor *would have bound to a different symbol* under language version 13 or earlier, precisely so this silent semantic drift cannot slip through unnoticed. The fix is the `@field` verbatim-identifier escape (or `this.field`), which forces the *identifier* interpretation:

```
public sealed class Plot
{
    private int field = 7;

    public int First
    {
        get => @field;              // unambiguous: the
                                   // legacy member named 'field' (== 7)
    }

    public int Capacity
    {
        get => field;               // the synthesized
                                   // backing field of Capacity (keyword)
        set => field = value < 0 ? 0 : value;
    }
}
```

`@field` always means “the identifier `field`”; bare `field` inside an accessor always means the backing field. Symmetrically, if you declare a *local or parameter* named `field` inside an accessor, C# 14 reports **error CS9273** (“`field` is a keyword within a property accessor. Rename the variable or use the identifier `@field` instead”) — that name is now reserved there — and you must rename it or escape its declaration as `@field`. (Note both diagnostics target *properties*; indexer and event accessors never treat `field` as a keyword, so legacy `field` identifiers there are untouched.) The guidance for any codebase upgrading to C# 14 is mechanical: enable the warning, search for the identifier `field` used inside property accessors, and either rename the legacy members or prefix every legacy use with `@`. Most teams never had a member named `field`; for those, the migration is a no-op. For those that did, the warning makes the hazard loud rather than silent.

Pitfall: silent semantic change on upgrade

The danger is not the compile error (CS9273 stops the build when a *local or parameter* named `field` is declared in an accessor), it is the *warning* case (CS9258) where the code still compiles but now means something different. A getter that read a legacy `field` member will, post-upgrade, read the synthesized backing field instead — usually returning `default` forever. Treat CS9258 as an error in your build (`<WarningsAsErrors>CS9258</WarningsAsErrors>` — CS9273 is already an error — or, better, `warnaserror` for the whole nullable/feature band) when migrating, so no occurrence slips through review.

Partial properties and partial indexers (C# 13)

Introduced in C# 13 (.NET 9, November 2024), **partial properties** and **partial indexers** extend the partial-member model — long available for methods — to properties. One declaration (the *defining* part) states the signature with accessor headers but no bodies; another (the *implementing* part) supplies the bodies. The defining part may add no `field`, no body; the implement-

ing part provides the logic. This exists almost entirely for **source generators**: the hand-written file declares intent, the generated file fills it in.

```
// --- Hand-written: Settings.cs ---
public partial class Settings
{
    // Defining declaration: signature only, no
    // bodies.
    public partial string Theme { get; set; }
}
```

```
// --- Generated: Settings.g.cs (emitted by a source
// generator) ---
public partial class Settings
{
    // Implementing declaration: bodies supplied. Note
    // this implementation
    // happens to use the C# 14 `field` keyword to
    // avoid a separate backing field.
    public partial string Theme
    {
        get => field ?? "Light";
        set
        {
            field = value;
            OnSettingChanged(nameof(Theme), value);
        }
    }

    partial void OnSettingChanged(string name, string
    value);
}
```

The defining and implementing parts must agree on accessor set, accessibility, type, and `ref`-ness; the implementing part is where bodies (and any `field` usage) live. Before C# 13, a generator that wanted to *augment* a property had to own the entire property in generated code, with no hand-written counterpart to reference; partial properties let the developer-facing declaration and the generated implementation coexist in the same member. Partial indexers follow the identical rule with `this[...]` syntax.

Note the version interplay: partial properties are **C# 13**; using `field` inside the implementing half is **C# 14**. A generator targeting a C# 13 consumer must still emit an explicit backing field; one targeting C# 14 can emit the leaner `field` form shown above. The two features are independent but compose.

INotifyPropertyChanged, reimaged with field

The MVVM `SetProperty` pattern is the textbook case where `field` removes an entire category of boilerplate. The classic `ObservableObject.SetProperty` from the MVVM Community Toolkit has the signature `protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string? propertyName = null)` — it compares, assigns by reference, raises `PropertyChanged` if the value changed, and returns whether it changed. Historically each property needed a named backing field to pass by `ref`:

```
// Pre-C# 14: explicit backing field required so it  
can be passed by ref.  
using CommunityToolkit.Mvvm.ComponentModel;  
  
public partial class PersonViewModel :  
    ObservableObject  
{  
    private string _name = "";  
    public string Name  
    {  
        get => _name;  
        set => SetProperty(ref _name, value);  
    }  
}
```

Because `field` is a real, `ref`-able storage location, C# 14 lets you pass it directly to `SetProperty`, eliminating the named field entirely while keeping full change-notification behavior:

```

using CommunityToolkit.Mvvm.ComponentModel;

// No 'partial' needed: we use SetProperty directly,
// not the [ObservableProperty] generator.
public class PersonViewModel : ObservableObject
{
    public string Name
    {
        get;
        set => SetProperty(ref field, value); // C#
        14: no manual backing field
    }

    public int Age
    {
        get;
        set => SetProperty(ref field, value);
    }
}

```

`SetProperty(ref field, value)` reads and writes the synthesized field by reference, compares via `EqualityComparer<T>.Default`, raises `PropertyChanged(nameof(Name))` (resolved by `[CallerMemberName]`) only on actual change, and returns the change flag if you need to chain side effects. This is strictly less code than the `[ObservableProperty]` source generator for simple cases — no `partial` requirement *for this reason*, no generator, no `field`-attribute incantations — while remaining fully transparent and steppable in the debugger. For properties that must trigger dependent updates, capture the return:

```

public class OrderViewModel : ObservableObject
{
    public decimal UnitPrice
    {
        get;
        set
        {
            if (SetProperty(ref field, value))
                OnPropertyChanged(nameof(Total)); //
                dependent computed property
        }
    }

    public int Quantity
    {
        get;
        set
        {
            if (SetProperty(ref field, value))
                OnPropertyChanged(nameof(Total));
        }
    }

    public decimal Total => UnitPrice * Quantity;
    // computed: no field, correct
}

```

If you do not want a toolkit dependency, the same shape works against a hand-rolled base. Note `Total` correctly uses an expression-bodied computed property with **no field** — a reminder that the feature is opt-in per property.

A worked example: a validated, observable, lazily-derived view model

The following pulls the chapter's threads together — validating `set`, `init` immutability, a lazy `field ??=` cache, change notification through `field`, the `@field` escape for a legitimately-named member, and a correctly *computed* property carrying no field.


```

#nullable enable
using System.ComponentModel;
using System.Runtime.CompilerServices;

public sealed partial class TradeTicket :
    INotifyPropertyChanged
{
    public event PropertyChangedEventHandler?
        PropertyChanged;

    private bool SetField<T>(ref T storage, T value,
        [CallerMemberName] string? name = null)
    {
        if
            (EqualityComparer<T>.Default.Equals(storage,
            value)) return false;
        storage = value;
        PropertyChanged?.Invoke(this, new
            PropertyChangedEventArgs(name));
        return true;
    }

    // Immutable, validated once at construction (C# 9
    // init + C# 14 field).
    public required string Symbol
    {
        get;
        init => field =
            value?.Trim().ToUpperInvariant() is { Length:
            > 0 } s
            ? s
            : throw new ArgumentException("Symbol
            required.", nameof(value));
    }

    // Mutable + observable, no manual backing field
    // (C# 14 field).
    public int Quantity
    {
        get;
        set
        {
            ArgumentOutOfRangeException.ThrowIfNegativeOrZ
            ero(value);
            SetField(ref field, value);
        }
    }

    public decimal Price
    {

```

```

    get;
    set
    {
        ArgumentOutOfRangeException.ThrowIfNegative(value);
        SetField(ref field, value);
    }
}

// Computed: NO field. Recalculated on each read,
// always correct.
public decimal Notional => Quantity * Price;

// Lazily computed, cached, null-resilient (C# 14
// field + C# 8 ??=).
public string Fingerprint =>
    field ??= $"{{Symbol}}:{{Quantity}}@{{Price}}:
    {{DateTime.UtcNow.Ticks:X}}";

// A legitimately-named member 'field' coexisting
// with the keyword.
private readonly string field = "TICKET";

public string Origin => @field; // @field
// escapes to the member, not the keyword
}

```

Every property here uses exactly the tool it needs: `init + field` for validated immutability, `set + field + SetField` for observable mutation, a plain computed member for `Notional`, a `field ??=` cache for `Fingerprint`, and `@field` to read the real member named `field`. There is not one hand-written `_backing` declaration among the observable and validated properties, yet every invariant is enforced and unforgeable.

One honest caveat about this example: because the class declares a member literally named `field` and uses the `field` keyword in its property accessors, the compiler raises **CS9258** on each bare-`field` use (`Symbol`, `Quantity`, `Price`, `Fingerprint`) — every one of them “would have bound to a different symbol” (the member) under C# 13. The code is correct and does exactly what the comments say, but it is *not* warning-clean. The legacy member is kept here only to exercise the `@field` escape; in real code you would rename it (`_kind`, say) and the warnings van-

ish. Treat the coexistence as a teaching device, not a pattern to emulate.

Note also `Fingerprint`’s cache is *stale by design* once `Quantity` or `Price` changes — a reminder that `field ??=` caching is correct only for values derived from immutable inputs, or where staleness is acceptable. For `Symbol` alone it would be safe; mixing in mutable `Quantity` / `Price` makes the cache a deliberate (and here, dubious) tradeoff you must reason about.

Performance, IL, and JIT notes

`field`-backed properties lower to the *same IL instruction sequences* as the equivalent property written with an explicit backing field — the accessor bodies emit identical `ldfld` / `stfld` against one instance field. (The metadata is not literally identical: the synthesized field gets an unspeakable, `[CompilerGenerated]` name like `<Name>k__BackingField` rather than the hand-chosen `_name`, but that is a name in the metadata table, not a runtime cost.) There is no extra indirection, no allocation beyond the one backing field that any stateful property requires. The synthesized field is a normal instance field; the JIT inlines trivial accessors (`get => field;`) to direct `ldfld`, exactly as it does for classic auto-properties, so the property-call abstraction vanishes at machine-code level. A validating setter is no slower than the hand-written equivalent because it *is* the hand-written equivalent after lowering.

GC behavior is likewise unchanged: a `field`-backed `string` property holds one reference, contributing one slot to the object’s layout. The feature is a pure source-level convenience; it neither adds nor removes work for the runtime. The only performance-relevant decision remains *semantic*: `field ??=` lazy caching trades memory (a permanent field, possibly `T?`) and a branch-on-read for amortized compute savings, and is unsafe under concurrency. If a property is cheap to compute and rarely read, prefer a plain computed member and pay the field’s eight bytes nowhere.

Choosing between the three property forms — a decision guide

You need...	Use	Storage
No behavior, just state	auto-property { get; set; }	one synthesized field
Behavior in one accessor over stored state	field -backed (auto + body)	one synthesized field
Behavior in both accessors over stored state	field -backed (two bodies)	one synthesized field
A value computed from other members	expression-bodied =>	none
Storage of a different type than the property (e.g. <code>Lazy<T></code> , an interlocked counter, a struct you mutate in place)	explicit field + full property	explicit field
A property a source generator must implement	partial property (C# 13); generator may use <code>field</code> internally	generator's choice
Validated, write-once immutability	<code>required</code> + <code>init</code> + <code>field</code>	one synthesized field
Change notification (MVVM)	<code>field</code> + <code>SetProperty(ref field, value)</code>	one synthesized field

The throughline: reach for `field` when the property has *state* and needs *behavior around that state*, and the state's type matches the property's type. Drop to an explicit field only when the backing storage must differ structurally from what the property exposes (`Lazy<T>` , `ImmutableInterlocked` targets, a value type you mutate by `ref` in many methods, or interop scenarios

needing a named, attributable field). Use a computed member whenever there is no state at all. And keep plain auto-properties for the genuinely trivial cases — adding `field` there is the noise this chapter has warned against twice.

When not to use `field`

- **Trivial properties.** `{ get; set; }` is shorter and clearer; `get => field; set => field = value;` is strictly worse.
- **Genuinely computed values.** A `field` there is a stale, never-needed allocation.
- **Cross-thread shared lazy state.** `field ??=` lacks publication guarantees; use `Lazy<T>` or `LazyInitializer`.
- **When the backing type must differ from the property type** in a structural way the resilience analysis can't paper over — an explicit field is honest and necessary.
- **Hot upgrade paths with legacy `field` members**, until you have audited CS9258 (and CS9273 for any `field` local/parameter) and applied `@field` — adopt deliberately, with warnings as errors, not by flipping `<LangVersion>` and hoping.

Used with discipline, `field` removes the largest remaining source of property boilerplate in C#: the trivial accessor you only wrote because the other one needed a body. It does not change what a property is — storage plus accessors — it just stops making you spell the storage out loud.

Sources: [The `field` contextual keyword — C# reference \(Microsoft Learn\)](#) · [The `field` contextual keyword — feature specification \(Microsoft Learn\)](#) · [What's new in C# 14 \(Microsoft Learn\)](#) · [C# compiler breaking changes since C# 13 \(Microsoft Learn\)](#) · [ObservableObject — MVVM Community Toolkit \(Microsoft Learn\)](#)

8 · Extension Members: Extension Everything (C# 14)

For seventeen years the only thing you could bolt onto a type you didn't own was a method whose first parameter wore a `this` modifier. C# 14 generalizes that single trick into a coherent feature: **extension blocks** that can declare extension properties, static extension members, and extension operators — including operators on types you have no source access to. This chapter dissects the new `extension(Type instance) { ... }` syntax, shows exactly how each member shape lowers to ordinary static IL, and works through where the feature earns its keep and where it quietly makes your codebase worse.

The classic extension method, and the walls it hit

The C# 3.0 extension method is a static method in a non-generic, non-nested static class whose first parameter is marked `this`. The compiler indexes such methods and lets you call them with instance-call syntax. The mechanism is pure syntactic sugar — the emitted IL is an ordinary static call.

```
public static class StringExtensions
{
    public static bool IsBlank(this string? value)
        => string.IsNullOrEmpty(value);
}

// Call site:
string? name = GetName();
if (name.IsBlank()) // sugar for
    StringExtensions.IsBlank(name)
    throw new ArgumentException(nameof(name));
```

Two properties of this design matter for the rest of the chapter. First, dispatch is entirely **static**: the method bound is chosen at compile time from the static type of the receiver, never from its runtime type. An extension method on `string` does not partici-

pate in virtual dispatch and cannot be overridden by a derived type. Second, the receiver can be `null` — `name.IsBlank()` on a null `name` does not throw an NRE, because no instance member is actually invoked; the null simply flows in as the first argument.

The walls were always about *member kind*. You could add methods and nothing else. You could not add a property, so `name.IsBlank` had to be `name.IsBlank()`. You could not add a static member, so `DateOnly.Tomorrow` was impossible — the best you could manage was a free-standing `DateOnlyExtensions.Tomorrow()` that read nothing like a member of `DateOnly`. And you could not add an operator, so adding `+` to a third-party `Vector2`-style struct meant a helper method `Add(a, b)` instead of `a + b`. Every workaround pushed call sites away from the natural member syntax the language otherwise rewards.

The extension block: `extension(Type receiver) { ... }`

C# 14 keeps the container exactly where it was — a **top-level, non-generic static class** — but lets that class hold one or more `extension` blocks. Each block names the **extended type** and, optionally, a **receiver parameter** that is in scope for every instance member inside the block.

```

public static class StringExtensions
{
    extension(string? value)           // receiver:
        'value' of type string?
    {
        public bool IsBlank =>
            string.IsNullOrEmpty(value);

        public string OrDefault(string fallback)
            => IsBlank ? fallback : value!;
    }
}

// Call site – note the property has no parentheses:
string? raw = GetName();
string name = raw.OrDefault("anonymous");
bool empty = raw.IsBlank;

```

The receiver name (`value`) behaves like `this` would inside a real instance member: it is the bound instance, and every instance member in the block reads it. Crucially, an extension **property** — `IsBlank` here — is brand new in C# 14 and is exactly what the classic `this` -parameter form could never express. It reads as a member access (`raw.IsBlank`) yet lowers to a static method call.

The placement rules are strict and worth memorizing, because the diagnostics if you break them are blunt:

- The containing class must be `static`, non-generic, and not nested. (Type parameters go on the `extension` block or member, never the class — see §“Generic extension blocks”.)
- A single `extension` block may declare any number of members, **as long as they share the same receiver**. Mix receivers and you split into multiple blocks.
- One static class may hold many `extension` blocks, each with its own receiver type and name. The block boundary is purely organizational; it introduces **no new scope** for signature uniqueness (more on that collision rule in §“Overload resolution”).

How extension members lower

Nothing about extension members reaches the CLR as a new concept. The compiler lowers every extension member to a static method on the containing class, and that is the only thing that exists at runtime. An expression-tree author, a reflection consumer, and the JIT all see ordinary statics.

An **extension property** lowers to a `get_` / `set_` accessor pair that takes the receiver as its leading argument. The C# 14 extension property `IsBlank` above produces, in effect:

```
// Conceptual lowering of the IsBlank extension  
property:  
public static bool get_IsBlank(string? value)  
    => string.IsNullOrEmpty(value);
```

An **instance extension method** lowers to exactly what a `this`-parameter method would have produced — which is why the two forms are binary and source compatible, and why you can migrate one to the other without rev'ing a public contract. A **static extension member** lowers to a static method whose name encodes the receiver *type* rather than taking a receiver instance. The practical upshot of the shared lowering: there is no per-call allocation, no boxing of value-type receivers beyond what a normal static call would do, and no extra indirection for the JIT to see through. Extension members are a front-end feature with zero runtime tax relative to the equivalent static call.

That static-call reality is also the source of the chapter's first and most important pitfall, so state it plainly now: **extension members are statically dispatched**. An extension property is not virtual, does not override anything, and does not participate in interface dispatch. If `IFoo` and a class `Foo : IFoo` both have an extension named `Tag` and you call `Tag` through an `IFoo` reference, you get the `IFoo` extension regardless of the runtime type. We return to this with a concrete failure in §“Pitfalls”.

A real extension property:

`ReadOnlySpan<char>.IsBlank`

The motivating example for extension properties is the one the `this`-parameter form physically could not write: a property on a `ref struct`. You cannot have a `this ReadOnlySpan<char>` property in C# 13 because there were no extension properties at all. C# 14 fixes both halves at once.

```
using System;

public static class SpanTextExtensions
{
    extension(ReadOnlySpan<char> span)
    {
        // C# 14 extension property on a ref struct receiver.
        public bool IsBlank => span.Trim().IsEmpty;

        // C# 14 extension property: length after trimming, no allocation.
        public int VisibleLength => span.Trim().Length;
    }
}

class Demo
{
    static void Main()
    {
        ReadOnlySpan<char> a = " ";
        ReadOnlySpan<char> b = " hi ";

        Console.WriteLine(a.IsBlank);           // True
        Console.WriteLine(b.IsBlank);           // False
        Console.WriteLine(b.VisibleLength);     // 2
    }
}
```

`ReadOnlySpan<char>.Trim()` and `.IsEmpty` are real BCL members (`MemoryExtensions.Trim` and `ReadOnlySpan<T>.IsEmpty`), so the body is genuine, allocation-free span work. The receiver is a `ref struct`, which means the lowered accessor takes a `ReadOnlySpan<char>` by value — fine, because a span is a cheap two-field struct (a managed ref plus a

length). No part of this captures the span on the heap, and because there is no `this` instance and no closure, `IsBlank` is exactly as cheap as calling `span.Trim().IsEmpty` inline. This is the canonical “extension properties pull their weight” case: span-centric APIs read far better as `span.IsBlank` than `SpanHelpers.IsBlank(span)`, and you pay nothing for the readability.

One C# 14 adjacency worth noting: first-class implicit `Span<T> / ReadOnlySpan<T>` conversions (also new in C# 14) mean a `Span<char>` argument binds to a `ReadOnlySpan<char>` receiver without a cast, so the same extension covers both span flavors at the call site.

Static extension members: adding

`DateOnly.Tomorrow`

When the only members in a block are static, the receiver parameter degenerates to just the **type** — you write `extension(DateOnly)` with no name, because there is no instance to bind. Static extension members let you add factory-style or constant-style members that *appear on the type itself*, called with `Type.Member` syntax.

```

using System;

public static class DateOnlyExtensions
{
    extension(DateOnly) // no receiver
        name: members are all static
    {
        // C# 14 static extension property – reads
        // like DateOnly.Today.
        public static DateOnly Tomorrow
            =>
            DateOnly.FromDateTime(DateTime.Now).AddDays(1)
            ;

        public static DateOnly Yesterday
            =>
            DateOnly.FromDateTime(DateTime.Now).AddDays(-1)
            ;

        // C# 14 static extension method.
        public static DateOnly NextWeekday(DayOfWeek
            day)
        {
            var d =
            DateOnly.FromDateTime(DateTime.Now);
            do { d = d.AddDays(1); } while
            (d.DayOfWeek != day);
            return d;
        }
    }
}

class Demo
{
    static void Main()
    {
        DateOnly t = DateOnly.Tomorrow;
        // static extension property
        DateOnly y = DateOnly.Yesterday;
        DateOnly mon =
        DateOnly.NextWeekday(DayOfWeek.Monday);
        Console.WriteLine($"{y} / {t} / {mon}");
    }
}

```

`DateOnly` lives in the BCL; you own none of it. Yet

`DateOnly.Tomorrow` now resolves to your static extension property, and the call site is indistinguishable from a built-in static. This is the feature’s headline trick — “extension everything” —

and it is genuinely useful for closing the gap where a framework type has `Today` but not `Tomorrow`, or `MinValue` but not your domain's `Epoch`.

It is also where the third pitfall lives, so flag it now: a reader who types `DateOnly.` and sees `Tomorrow` in IntelliSense has no immediate signal that it is *yours* and not Microsoft's. Static extension members are the easiest way in C# 14 to obscure where a type's API actually lives. Use them where the addition is genuinely “this should have been on the type” (a missing factory), not to scatter app-specific helpers across framework types.

Extension operators, including on types you don't own

The most ambitious new shape is the **extension operator**. You can define `operator +`, `operator *`, comparison operators, and even user-defined compound assignment operators on a type you do not control, provided the extended type appears as a parameter. Operators are static by nature, so they live in a block whose receiver is just the type (or a named receiver if the block also holds instance members).

`System.Numerics.Vector2` ships with its own `+`, so to demonstrate *adding* an operator we'll use a small struct the example owns the absence of operators on — a 2-D integer point with no operators — and add `+` and `*=` from outside:

```

using System;

// Imagine this comes from a library you can't edit;
// it has NO operators.
public readonly struct GridVec(int x, int y)
{
    public int X { get; } = x;
    public int Y { get; } = y;
    public override string ToString() => $"({X}, {Y})";
}

public static class GridVecOperators
{
    extension(GridVec)
    {
        // C# 14 extension operator: binary '+', both
        // operands explicit, static.
        public static GridVec operator +(GridVec a,
            GridVec b)
            => new(a.X + b.X, a.Y + b.Y);

        public static GridVec operator *(GridVec v,
            int scalar)
            => new(v.X * scalar, v.Y * scalar);
    }
}

class Demo
{
    static void Main()
    {
        GridVec a = new(1, 2), b = new(3, 4);
        GridVec sum = a + b; // extension
        // operator +
        GridVec twice = a * 2; // extension
        // operator *
        Console.WriteLine($"{sum} {twice}"); // (4,
        // 6) (2, 4)
    }
}

```

The declaration rules (from the C# 14 extension-operators specification) are precise:

- A static extension operator for extended type `T` must take **at least one parameter whose type is identity-convertible to `T`**. You cannot smuggle in `operator +(int,`

`int`) under `extension(GridVec)` — the extended type must actually appear.

- Operators that the language requires in pairs must both be present, and the two halves may live in **separate extension blocks in the same static class**. `operator ==` requires `operator !=`; `operator <` requires `operator >`; and so on.
- Extension operators cannot use `abstract`, `virtual`, `override`, or `sealed`.
- An extension operator is only considered when **no pre-defined or non-extension user-defined operator applies**. So you cannot use an extension operator to *override* a type's real `+`; if `Vector2` already defines `+`, your `extension(Vector2) operator +` is dead code — the built-in always wins. This is the operator-collision pitfall in §“Pitfalls”.

C# 14 also introduced **user-defined compound assignment operators** (the `operator +=` form, a void-returning instance operator that mutates in place), and they may be declared as extension members too. The rules are stricter because they mutate the receiver:

```
public static class GridVecCompound
{
    // Value-type receiver mutated in place =>
    // receiver parameter must be 'ref'.
    extension(ref GridVec self)
    {
        // C# 14 user-defined compound assignment
        // operator, as an extension.
        public void operator +=(GridVec other)
            => self = new(self.X + other.X, self.Y +
                other.Y);
    }
}
```

Note the shape: a compound assignment operator has **no static modifier**, returns `void`, takes exactly one parameter (the right-hand operand), and mutates the receiver. Because `GridVec` is a value type, the receiver must be a `ref` parameter

(`extension(ref GridVec self)`); for a reference-type receiver it would be a plain value parameter, and the receiver type may not be an unconstrained type parameter. The payoff is the same as for in-type compound operators: `a += b` can mutate `a` in place rather than allocating a fresh value and reassigning. For a tiny `readonly struct` the win is negligible, but for a larger mutable struct or a buffer-backed type it removes a copy. When both a compound `+=` and a binary `+` are in scope, `a += b` prefers the compound form and falls back to expanding to `a = a + b` only if no compound operator exists.

Generic extension blocks

When the receiver itself is generic, put the type parameter on the `extension` block: `extension<T>(IEnumerable<T> source)` . Member-only type parameters (ones the receiver doesn't mention) go on the member. The rule is exact: a type parameter used in the receiver goes on the block; a type parameter distinct from the receiver's goes on the member; you may not declare the same parameter in both places.

Here is a generic block over `IEnumerable<T>` that exposes both a method and a `Count`-like property — the latter being the kind of member the `this`-parameter form could never write:


```

using System;
using System.Collections.Generic;
using System.Linq;

public static class SequenceExtensions
{
    extension<T>(IEnumerable<T> source)
    {
        // C# 14 extension property over a generic
        // receiver.
        // A materialized count that reads like a
        // member, computed on demand.
        public int FastCount
            => source is ICollection<T> c ? c.Count :
            source.Count();

        // Member-only type parameter TKey goes on the
        // method, not the block.
        public IEnumerable<T> DistinctBy2<TKey>
            (Func<T, TKey> keySelector)
        {
            var seen = new HashSet<TKey>();
            foreach (var item in source)
                if (seen.Add(keySelector(item)))
                    yield return item;
        }

        public bool IsEmpty => !source.Any();
    }
}

class Demo
{
    static void Main()
    {
        int[] xs = [3, 1, 4, 1, 5, 9, 2, 6];
        Console.WriteLine(xs.FastCount);    // 8
        // (ICollection fast path)
        Console.WriteLine(xs.IsEmpty);      // False

        var unique = xs.DistinctBy2(n => n %
            3).ToArray();
        Console.WriteLine(string.Join(",", unique));
        // 3,1,5
    }
}

```

FastCount deserves a word of caution that has nothing to do with the syntax and everything to do with *naming a property*. A

property *looks* free — readers assume `O(1)`, no side effects, idempotent. `source.Count()` on a non-`ICollection` sequence enumerates the whole thing, which for a lazy or one-shot `IEnumerable` is `O(n)` and may even consume the sequence. Exposing that behind a property named `FastCount` is a lie the call site can't see. The `ICollection<T>` fast path mitigates it, but the general lesson holds: only promote an enumerable operation to an extension *property* when it is genuinely cheap and side-effect-free. When in doubt, keep it a method so the parentheses warn the reader that work happens.

Coexistence with `this`-parameter extension methods

You do not have to choose. The classic form and the new form can live in the **same static class**, and because they lower identically, callers can't tell which one they're using. A realistic migration keeps stable, widely-called methods as-is and adds new property/static/operator members in `extension` blocks:

```

using System;

public static class TemperatureExtensions
{
    // Legacy this-parameter extension method –
    // unchanged, still binary compatible.
    public static double ToFahrenheit(this double
        celsius)
        => celsius * 9.0 / 5.0 + 32.0;

    // New C# 14 members for the same conceptual
    // receiver, side by side.
    extension(double celsius)
    {
        public bool IsFreezing => celsius <= 0.0;
        // extension property
        public double ToKelvin() => celsius + 273.15;
        // extension method, new form
    }
}

class Demo
{
    static void Main()
    {
        double c = -4.0;
        Console.WriteLine(c.ToFahrenheit()); // 24.8
        // (legacy form)
        Console.WriteLine(c.IsFreezing);     // True
        // (new extension property)
        Console.WriteLine(c.ToKelvin());     // 269.15
    }
}

```

The one hard rule across both forms is signature uniqueness within the class. The `extension` block introduces **no scope**: every member declared in a single static class — across all its `extension` blocks and all its legacy `this` -methods — must have a unique generated signature. For instance members the generated signature includes the receiver parameter; for static members it includes the receiver type. So you cannot declare `IsFreezing` twice with the same receiver type even if they're in different blocks, and you cannot have both a legacy `ToKelvin(this double)` and a new-form `ToKelvin()` over `double` in the same class — they collide.

Migrating a helper class — call sites before and after

The clearest argument for the feature is a side-by-side. Here is a conventional bag of `this`-parameter string helpers:

```
// BEFORE — C# 13 static helper class.
public static class HtmlHelpers
{
    public static string Truncate(this string s, int
        max)
        => s.Length <= max ? s : s[..max] + "...";

    public static bool LooksLikeEmail(this string s)
        => s.Contains('@') && s.IndexOf('@') ==
            s.LastIndexOf('@');

    // Awkward: a "static-ish" helper that has no good
    // home on string.
    public static string EmptyMarker() => "-"; // em
    dash
}
```

The same surface re-expressed with C# 14 extension members, promoting the predicate to a property and the marker to a *static* extension property on `string` itself:

```
// AFTER — C# 14 extension block.
public static class HtmlHelpers
{
    extension(string s)
    {
        public string Truncate(int max)
            => s.Length <= max ? s : s[..max] + "...";

        public bool LooksLikeEmail
            => s.Contains('@') && s.IndexOf('@') ==
                s.LastIndexOf('@');
    }

    extension(string)
    {
        public static string EmptyMarker => "-";
    }
}
```

The call sites tell the story:

```
// BEFORE
if (input.LooksLikeEmail())           // method-call
    Send(input);                      // parens on a predicate
var label = name.Truncate(20);
var blank = HtmlHelpers.EmptyMarker(); // free
                                         // function, reads nothing like 'string'

// AFTER
if (input.LooksLikeEmail)             // reads as a
    Send(input);                      // property – no parens
var label = name.Truncate(20);         // unchanged
var blank = string.EmptyMarker;        // appears to
                                         // belong to string
```

`LooksLikeEmail` losing its parentheses is a real ergonomic gain — predicates read like the boolean properties they conceptually are. `string.EmptyMarker` is the double-edged one: cleaner than `HtmlHelpers.EmptyMarker()`, but now an app-specific constant masquerades as a member of `string`. That is the readability-versus-honesty trade you make every time you reach for a static extension member.

Overload resolution, ambiguity, and discoverability

Extension members slot into the same resolution machinery as classic extension methods, with the same priority: the compiler first looks for an applicable **declared member** on the receiver's type — an instance member for instance access, a static member for `Type.Member` access — and only if none is found does it consider **extensions**, searching scope by scope from the innermost namespace outward, pulling in `using`-imported static classes at each level. The first scope that yields at least one applicable candidate wins; resolution does not keep climbing once a scope produces a match. This means a real member *always* shadows an extension of the same name and kind — if the BCL adds a static `DateOnly.Tomorrow` in a future release, your static extension property silently stops being called, with no error, be-

cause the type's own static member now wins. The same trap applies to instance access: a real `string.SomeProperty` would shadow your `extension(string) { public ... SomeProperty }`.

When two extensions are equally applicable in the same scope, you get an ambiguity error and must **disambiguate by calling through the static class**. For extension members the disambiguation forms are specific:

```
// Two static classes both define an extension
// property IsBlank for string:
string? s = " ";

// Ambiguous if both are in scope:
// bool b = s.IsBlank;           // CS error:
//                               // ambiguous

// Disambiguate by invoking the generated accessor
// through the owning class,
// passing the receiver as the leading argument:
bool b1 = StringExtensions.get_IsBlank(s);
bool b2 = OtherStringExtensions.get_IsBlank(s);
```

For an ambiguous *instance extension method* you call `OwningClass.Method(receiver, args...)`; for a *static extension member* you call the generated zero-receiver-argument static; and for an *extension property* you call the `get_ / set_` accessor with the receiver as the first argument, as above. These `get_`-prefixed accessors are normally invisible, but they are exactly what the compiler emits, and they are your escape hatch when names collide. Operators disambiguate differently — there is no syntax to qualify `a + b` to a specific extension operator, so genuinely ambiguous extension operators in the same scope are simply a compile error you must resolve by removing or relocating one of them.

Discoverability is the quiet cost. Extension members only appear in IntelliSense when their containing static class is imported into the current scope (via a matching `using`). An extension property on `ReadOnlySpan<char>` defined in a namespace you haven't imported is invisible; the reader sees neither the member nor a

hint that it exists. This is identical to classic extension-method discoverability, but it now applies to things that *look* like properties and static members, which raises the surprise factor when a familiar-looking `DateOnly.Tomorrow` compiles in one file and not in another.

A worked example: an allocation-aware fluent CSV field reader

Pulling the threads together — extension property, generic extension block, static extension member, and span-aware design — here is a small, allocation-conscious utility for walking comma-separated fields over a `ReadOnlySpan<char>` without ever allocating a `string[]` or `List<string>`:

```

using System;

public static class CsvSpanExtensions
{
    extension(ReadOnlySpan<char> line)
    {
        // Extension property: is this line structurally empty?
        public bool IsEmptyRow => line.Trim().IsEmpty;

        // Extension property (explicit get accessor): count fields without allocating.
        public int FieldCount
        {
            get
            {
                if (line.IsEmpty) return 0;
                int count = 1;
                foreach (char c in line)
                    if (c == ',') count++;
                return count;
            }
        }

        // Extension method: get the Nth field as a sliced span (no copy).
        public ReadOnlySpan<char> Field(int index)
        {
            ReadOnlySpan<char> rest = line;
            for (int i = 0; ; i++)
            {
                int comma = rest.IndexOf(',');
                ReadOnlySpan<char> cell = comma < 0 ?
rest : rest[..comma];
                if (i == index) return cell.Trim();
                if (comma < 0) return
ReadOnlySpan<char>.Empty;
                rest = rest[(comma + 1)..];
            }
        }
    }

    extension(ReadOnlySpan<char>)
    {
        // Static extension property: the conventional field delimiter.
        public static char Delimiter => ',';
    }
}

```



```

class Demo
{
    static void Main()
    {
        ReadOnlySpan<char> row = " alice , 42 , prague
        ";

        Console.WriteLine(row.IsEmptyRow);           //
        False
        Console.WriteLine(row.FieldCount);           // 3
        Console.WriteLine(row.Field(0).ToString()); //
        alice
        Console.WriteLine(row.Field(2).ToString()); //
        prague

        Console.WriteLine(ReadOnlySpan<char>.Delimiter
        ); // ,
    }
}

```

Every operation here is allocation-free until you deliberately materialize a field with `.ToString()`. `Field` returns a *slice* of the original span — a windowed view, not a copy — so parsing a million-row file touches the heap only for the rows you actually convert to strings. The `IsEmptyRow` and `FieldCount` extension properties read naturally, and `ReadOnlySpan<char>.Delimiter` adds a discoverable constant to the type. None of this was expressible with `this`-parameter methods: you could not put a property or a static member on a `ref struct` before C# 14. The design also illustrates the allocation-aware fluent style the feature enables — return spans, expose cheap properties, keep the heap untouched until the caller asks for a `string`.

A JIT/codegen note: because all of these lower to plain static methods taking a `ReadOnlySpan<char>` by value, and a span is a two-field by-ref struct, there is no boxing and no defensive copy beyond the normal struct-by-value argument pass. The `foreach (char c in line)` over a `ReadOnlySpan<char>` uses the span's struct enumerator, which the JIT routinely inlines into a bounds-checked loop. Performance is identical to having written the static helpers by hand; the extension syntax buys readability at zero runtime cost.

Pitfalls

Expecting static dispatch to be virtual dispatch.

Extension members bind on the *static* type of the receiver, full stop. This bites hardest with interfaces:

```
public interface IShape { }
public sealed class Circle : IShape { }

public static class ShapeExtensions
{
    extension(IShape shape) { public string Kind =>
        "shape"; }
    extension(Circle circle) { public string Kind =>
        "circle"; }
}

// ...
IShape s = new Circle();
Console.WriteLine(s.Kind); // "shape" - bound on
                           static type IShape, NOT "circle"

Circle c = new Circle();
Console.WriteLine(c.Kind); // "circle" - different
                           static type, different binding
```

The runtime type `Circle` is irrelevant; `s` is typed `IShape`, so the `IShape` extension wins. An extension property can never override or be polymorphic over a real member or another extension. If you need polymorphism, you need an actual virtual/interface member on the type — extensions are the wrong tool.

Colliding with a type's real operators. An extension operator is only consulted when no predefined or non-extension user-defined operator applies. So adding `extension(decimal) operator +` does nothing useful — `decimal` already has `+`, and the built-in always wins. Worse is the *near*-collision: an extension operator whose operands differ just enough to be selected in some expressions and shadowed in others produces code where `a + b` means different things depending on inferred operand types. If a type already has the operator you want, you have no leverage; if it doesn't, add the operator deliberately and

add its required pair (`==` with `!=` , `<` with `>`) or you'll get a compile error.

Obscuring where the API lives. This is the soft, cumulative failure. `DateOnly.Tomorrow` , `string.EmptyMarker` , `ReadOnlySpan<char>.Delimiter` — each reads beautifully and each erases the boundary between framework and your code. A new team member greps the BCL for `Tomorrow` , finds nothing, and is confused. Reserve static extension members and extension operators for cases where the addition is *obviously* a missing piece of the type's natural API; route app-specific logic through named helper types or instance members where the ownership is legible. The feature's name is “extension everything,” but the discipline is *not* extending everything.

Part III — Generics, Math, and Abstraction

9 · Generics Deep Dive: Variance, Constraints, and Specialization

Generics are the single most consequential feature in the C# type system for performance and abstraction, and almost everyone underestimates how much behavior is decided by the runtime rather than the language. This chapter works through the constraint catalog, the rules of declaration-site variance, the way the CLR shares one machine-code instantiation across all reference types while specializing each value type, and the techniques — `allows ref struct`, `EqualityComparer<T>.Default`, CRTP, and `typeof(T)` hot paths — that let you write generic code with zero boxing and zero allocations.

The constraint catalog

A type parameter without constraints is treated by the compiler as `object` with no `new()`, no operators, and a nullability state of “could be either.” Constraints buy you capabilities. Here is the full catalog, in the order the compiler requires them to appear:

```

// Primary constraint (at most one): class, class?,
// struct, unmanaged, notnull,
// a base class, or – C# 13 – the anti-constraint
// allows ref struct.
// Then: interface constraints. Then: new() (must be
// last).

void A<T>() where T : class { }           //
// reference type; T is non-nullable ref
void B<T>() where T : class? { }          //
// reference type; nullability oblivious
void C<T>() where T : struct { }           // non-
// nullable value type (Nullable<T> excluded)
void D<T>() where T : unmanaged { }        // struct
// with no managed references, recursively
void E<T>() where T : notnull { }          // value
// type OR non-nullable reference type
void F<T>() where T : Stream { }           // base-
// class constraint
void G<T>() where T : IComparable<T> { }   //
// interface constraint (can be self-referential)
void H<T>() where T : new() { }            // public
// parameterless constructor
void I<T>() where T : Stream, IDisposable, new() { }
// combination, in required order

```

Several of these interact in non-obvious ways:

- `struct` implies `notnull` and forbids `Nullable<T>` (you cannot pass `int?` for a `struct`-constrained `T`). It also gives the compiler permission to use `default(T)` cheaply and to avoid null checks.
- `unmanaged` (introduced in C# 7.3) implies `struct` and additionally guarantees the type contains no managed references anywhere in its transitive layout. That is what makes `stackalloc T[n]`, `sizeof(T)`, and pointer arithmetic on `T*` legal in unsafe contexts. (Strictly, “unmanaged” is about the absence of GC references, not “blittable” in the marshalling sense — `bool` and `char` are unmanaged but not blittable for interop. C# 12’s `[InlineArray]` structs are a common consumer of this constraint, but did not change its meaning.)
- `notnull` (C# 8) is purely a nullable-reference-types contract. It does **not** restrict you to value types; it forbids passing a nullable reference type where the compiler can prove

nullability. Violations are warnings, not errors, because NRT is a flow-analysis feature, not a CLR guarantee.

- `new()` cannot be combined with `struct` (every struct already has an implicit parameterless constructor, so the combination is redundant and disallowed). It compiles down to a call to `Activator.CreateInstance<T>()` for reference types — cheap now, but historically a reflection cost worth knowing about.

The constraint you cannot express is “has operator `+`.” That gap is what generic math (`INumber<T>` , C# 11 / .NET 7) closes via interface constraints with `static abstract` members — covered in the next chapter, but it is the reason `Sum<T>` later in this chapter constrains on `INumber<T>` rather than wishing for an operator constraint.

`allows ref struct` : the anti-constraint (C# 13)

Every constraint listed above *narrows* the set of permitted type arguments. C# 13 introduced the first *anti-constraint*, `allows ref struct`, which *widens* it: it tells the compiler that `T` may be a `ref struct` such as `Span<T>` or `ReadOnlySpan<T>`. Without it, ref structs are barred from being type arguments at all, because the CLR cannot prove their stack-only lifetime rules hold once a value is hidden behind an unconstrained `T`.

```
// C# 13 / .NET 9+. T may now be Span<T>,
//   ReadOnlySpan<T>, or any ref struct –
// as well as ordinary types. The compiler enforces
//   ref-struct lifetime rules on T.
static bool AnyMatch<T, TPredicate>(T source,
    TPredicate predicate)
    where T : allows ref struct
    where TPredicate : IPredicate, allows ref struct
{
    // Inside the method body, T is treated AS IF it
    //   were a ref struct:
    // you cannot box it, store it in a field, capture
    //   it in a lambda, or
    // use it as another type argument that lacks
    //   'allows ref struct'.
    return predicate.Test(source); // infers Test<T>
    //   (source) – the interface method's
    //                               // own type
    //   parameter must also allow ref structs.
}

interface IPredicate { bool Test<TValue>(TValue value)
    where TValue : allows ref struct; }
```

The crucial restriction: inside a method whose `T` has `allows ref struct`, the compiler must assume `T` *could* be a ref struct, so it forbids every operation a ref struct forbids — boxing, async captures, lambda captures, array element storage, and passing `T` to another generic API that does **not** itself declare `allows ref struct`. The anti-constraint is viral: it propagates down the call graph. It also cannot be combined with `class` or `class?`, since a ref struct is never a reference type.

The payoff is real BCL surface. `Span<T>` enumeration, `MemoryExtensions`, and several `System.Numerics.Tensors.TensorPrimitives` generic overloads use this so that a stack-allocated span can flow through a generic algorithm with no heap escape.

A boxing-free span algorithm with `where T : unmanaged`

Here is a complete, compilable kernel that sums a span of any unmanaged numeric-like type without a single allocation or box. `MemoryMarshal.Cast<TFrom, TTo>` itself only constrains both

parameters to `struct` and performs its “no managed references” check at *runtime* (throwing `ArgumentException` if either type contains GC references). The `unmanaged` constraint on `T` is what *statically* guarantees that runtime check always passes, and is what makes the reinterpret to `Vector<T>` and pointer-free SIMD meaningful in the first place.

```

using System;
using System.Numerics;
using System.Runtime.InteropServices;

static class Kernels
{
    // T : unmanaged guarantees no managed references
    //   in T's layout, so Vector<T>
    //   and reinterpret casts are sound. No boxing: T
    //   is a value type, never crosses to object.
    public static T SumUnmanaged<T>(ReadOnlySpan<T>
        data)
        where T : unmanaged, INumber<T>
    {
        T total = T.Zero;
        int i = 0;

        if (Vector.IsHardwareAccelerated &&
            Vector<T>.IsSupported && data.Length >=
            Vector<T>.Count)
        {
            var acc = Vector<T>.Zero;
            // Reinterpret the span as a span of SIMD
            // vectors – zero-copy.
            ReadOnlySpan<Vector<T>> vectors =
            MemoryMarshal.Cast<T, Vector<T>>(data);
            foreach (var v in vectors)
                acc += v;
            total = Vector.Sum(acc);
            i = vectors.Length * Vector<T>.Count;
        }

        for (; i < data.Length; i++)
            total += data[i];

        return total;
    }
}

// Usage – note the span never leaves the stack and no
// T is ever boxed:
Span<int> nums = stackalloc int[1024];
for (int k = 0; k < nums.Length; k++) nums[k] = k;
int s = Kernels.SumUnmanaged<int>(nums); // 523776

```

`INumber<T>` (C# 11 / .NET 7) supplies `T.Zero` and operator `+` as static abstract interface members, so the addition is a *constrained call* (more on that below) rather than a virtual dispatch through a boxed instance. `MemoryMarshal.Cast<T,`

`Vector<T>>` then reinterprets the bytes with no copy. Note the cast's length semantics: when `sizeof(Vector<T>)` does not evenly divide `data.Length * sizeof(T)`, the resulting `ReadOnlySpan<Vector<T>>` simply drops the trailing partial vector — that is exactly why the scalar `for` loop after the SIMD block resumes at `i = vectors.Length * Vector<T>.Count` to mop up the remainder. Drop the `unmanaged` constraint down to a bare `struct` and the code still compiles, but a `T` carrying managed references (say a struct with a `string` field) would now throw `ArgumentException` from `Cast` at runtime — and `Vector<T>` is only ever supported for primitive numeric `T` anyway.

How the runtime shares and specializes generic code

This is the single most misunderstood fact about .NET generics, so be precise about it: **the JIT generates one shared instantiation of generic code for all reference-type arguments, and a separate specialized instantiation for each distinct value-type argument.**

`List<string>`, `List<object>`, and `List<Stream>` all execute the *same* native code — the “canonical” instantiation, parameterized by a hidden type handle (`MethodTable*`) passed at runtime for the operations that actually need it. This is possible because every reference is one pointer-width slot with identical GC semantics. `List<int>`, `List<double>`, and `List<MyStruct>` each get their *own* native code, because their element sizes, copy semantics, and GC layouts differ.

```
// Reference types: shared code. Both calls run
// identical JIT output.
var a = new List<string>();
var b = new List<Uri>();

// Value types: each is a distinct, separately-JITted
// instantiation.
var c = new List<int>();           // own native code, no
// boxing of elements
var d = new List<DateTime>();     // own native code
```

The performance consequences are exactly what you want for value types (no boxing, inlined struct copies, devirtualized constrained calls) and a subtle cost ceiling for reference types (one extra dictionary lookup for `typeof(T)` or `new T()` in the shared instantiation — usually negligible, occasionally visible in tight loops via runtime “generic dictionary” lookups).

Pitfall — “generic code is always specialized per type.”

It is not. For reference types you get *one* body. So a micro-optimization that “works because the JIT can see T is sealed” will not fire in the shared instantiation, because the shared body has no idea which reference type it is running as.

Devirtualization of an interface call on a reference-type `T` generally cannot happen at JIT time; it relies on dynamic PGO or guarded devirtualization at runtime. Only value-type `T` gives you guaranteed monomorphic, inlinable calls.

Constrained calls, devirtualization, and avoiding boxing

When you call an interface method on a value-type `T` constrained to that interface, the C# compiler emits the `constrained.` IL prefix before `callvirt`. For a value type this resolves the method *without boxing*: the JIT dispatches directly to the struct’s implementation and can inline it. For a reference type the same IL boxes only if the method is inherited from `object` and not overridden — otherwise it is an ordinary virtual call.

```

interface IShape { double Area(); }

readonly struct Circle(double r) : IShape
{
    public double Area() => Math.PI * r * r;
}

// T : IShape, struct → constrained call, NO boxing,
// JIT can inline Circle.Area().
static double TotalArea<T>(ReadOnlySpan<T> shapes)
    where T : struct, IShape
{
    double sum = 0;
    foreach (ref readonly T s in shapes)
        sum += s.Area(); // constrained callvirt →
        // direct, inlinable, allocation-free
    return sum;
}

// Contrast: the non-generic version boxes every
// element on the IShape cast.
static double TotalAreaBoxing(ReadOnlySpan<IShape>
    shapes) // each Circle already boxed to fit
{
    double sum = 0;
    foreach (var s in shapes) sum += s.Area(); //
    // virtual call through boxed instance
    return sum;
}

```

The `struct` constraint is what makes the difference. Drop it to a bare `where T : IShape` and the compiler must assume `T` could be a reference type, so it routes through the shared instantiation and the `Area()` call becomes a genuine virtual dispatch that the JIT cannot reliably inline.

Pitfall — hidden boxing from `default(T)` and comparisons.

If `T` is an *unconstrained* type parameter and you write `if (value.Equals(default(T)))` or `if (value == null)`, you can silently box. `default(T)` for an unconstrained `T` is `null` when `T` is a reference type and a zeroed value when it is a struct — fine — but `value.Equals(other)` on an unconstrained `T` calls `object.Equals` and boxes both operands when `T` is a value type. The fix is `EqualityComparer<T>.Default.Equals(value, other)`,

which never boxes (see below). Also avoid `value == null` on unconstrained `T`; prefer `value is null`, which the compiler lowers correctly for both kinds.

default(T), EqualityComparer<T>.Default, and allocation-free comparison

`EqualityComparer<T>.Default` is the canonical way to compare two `T` without boxing and without knowing whether `T` implements `IEquatable<T>`. The runtime caches a single comparer instance per `T`, and for value types that implement `IEquatable<T>` it returns a specialized comparer whose `Equals` is a non-boxing, inlinable constrained call.

```
// Allocation-free "contains" for any T, value or
// reference.
static bool Contains<T>(ReadOnlySpan<T> span, T
    target)
{
    var cmp = EqualityComparer<T>.Default; // cached
    // singleton, no allocation
    foreach (ref readonly T item in span)
        if (cmp.Equals(item, target)) // never
            // boxes, even for value types
            return true;
    return false;
}
```

Why prefer this over `item.Equals(target)`? For an unconstrained value-type `T` that does *not* implement `IEquatable<T>`, `item.Equals(target)` resolves to the virtual `ValueType.Equals(object)`, which **boxes the target argument on every call** (it is typed as `object`). Worse, `ValueType.Equals` only takes a fast byte-by-byte path when the struct's fields are all bit-comparable value types; if any field is a reference type (or has padding holes / a custom `Equals`), it falls back to a **reflection-driven field-by-field** comparison — catastrophic in a hot loop. `EqualityComparer<T>.Default` instead inspects `T` once; if it implements `IEquatable<T>` it returns a `GenericEqualityComparer<T>` whose `Equals` is a non-boxing constrained call; otherwise it returns an

`ObjectEqualityComparer<T>` that still boxes but is at least cached. For reference types it dispatches through the shared instantiation to the type's `Equals` / `GetHashCode`. This is exactly how `Dictionary<TKey, TValue>` and `List<T>.IndexOf` work internally.

`default(T)` itself is allocation-free — it never calls a constructor; it produces the all-zeroes value or `null`. The trap is only in what you do with it afterward.

Covariance and contravariance (C# variance, declaration-site)

C# supports *declaration-site* variance on **interfaces and delegates only**, using `out` (covariant — `T` appears only in output positions) and `in` (contravariant — `T` appears only in input positions). Variance lets a more-derived or less-derived generic instance substitute for another along the assignment direction.

```

// Covariant producer: out T → T only in output
// positions (returns, get-only).
interface IProducer<out T>
{
    T Produce();
    // void Consume(T item); // ✗ would not
    // compile: T in an input position
}

// Contravariant consumer: in T → T only in input
// positions (parameters).
interface IConsumer<in T>
{
    void Consume(T item);
    // T Produce(); // ✗ would not
    // compile: T in an output position
}

sealed class StringBox : IProducer<string>
{
    private readonly string _v = "hi";
    public string Produce() => _v;
}

sealed class ObjectSink : IConsumer<object>
{
    public void Consume(object item) =>
        Console.WriteLine(item);
}

// Covariance: a producer of string IS-A producer of
// object (widening on output).
IProducer<object> po = new StringBox(); // ✓
    implicit: string Produce() satisfies object
    Produce()
object o = po.Produce();

// Contravariance: a consumer of object IS-A consumer
// of string (narrowing on input).
// No cast needed – the conversion is implicit and
// statically safe: anything that can
// consume *any* object can certainly consume a
// string.
IConsumer<string> cs = new ObjectSink(); // ✓
    implicit contravariant conversion
cs.Consume("hello"); //
    forwards a string to Consume(object)

```

Here is the compile-time variance violation the chapter must show, and *why* it is unsafe:


```
// ❌ Does NOT compile. The mnemonic: out = output-
// only, in = input-only.
interface IBroken<out T>
{
    void Consume(T item); // CS1961: Invalid
                          // variance: 'T' must be contravariantly
                          // valid on
                          // 'IBroken<T>.Consume(T)'. 'T' is covariant.
}
```

The compiler rejects this because covariance promises you can treat `IProducer<string>` as `IProducer<object>`. If the interface also had `Consume(T)`, an `IProducer<object>` reference could call `Consume(new object())` on an underlying string-only implementation — a type hole. The `in / out` rules are precisely the positions that make such substitution provably safe. Real-world payoff: `IEnumerable<out T>` lets `IEnumerable<string>` flow where `IEnumerable<object>` is expected; `Action<in T>` and `IComparer<in T>` let an `IComparer<object>` sort strings.

Pitfall — array covariance is unsafe, unlike interface variance. Arrays are covariant in C# (`string[]` is assignable to `object[]`), but that covariance is **not type-safe** and is checked at *runtime*, not compile time. Interface variance with `in / out` is checked at compile time and is always sound.

```
object[] arr = new string[3]; // legal – array
                             // covariance, compiles fine
arr[0] = "ok";                // fine
arr[1] = 42;                  // ⚠️ compiles, then
                             // throws ArrayTypeMismatchException at runtime

IEnumerable<object> seq = new string[3]; // interface
                                         // covariance – fully safe
// No write surface exists on IEnumerable<out T>, so
// no runtime hole is possible.
```

Every array store on a covariant array reference pays a hidden runtime type check, and a wrong type throws `ArrayTypeMismatchException`. This is why `List<T>` is invariant (no such hole) and why you should pass

`ReadOnlyList<T>` / `IEnumerable<T>` rather than `T[]` when you want covariance without the landmine.

Generic type inference, and where it fails

C# infers type arguments from method-argument types so you rarely write them explicitly. The algorithm gathers *bounds* from each argument, then finds a best common type. It is powerful but has hard edges you will hit:

```
static T Identity<T>(T x) => x;
var a = Identity(42);           // T = int, inferred
                                from the argument

static TResult Map<TSource, TResult>(TSource s,
    Func<TSource, TResult> f) => f(s);
var b = Map(3, x => x.ToString()); // TSource=int
                                inferred; TResult=string from lambda body
```

Where inference fails:

1. **Return-type-only inference.** C# never infers a type argument from the *expected return type* of the call site. `T Parse<T>(string s)` cannot infer `T` from `int x = Parse("5");` — you must write `Parse<int>("5")`. (Contrast with target-typed `new()` and collection expressions, which are a separate mechanism.)
2. **Inference through nested generic constraints.** Constraints participate in *checking*, never in *inferring*. The compiler infers from arguments first, then verifies constraints; it will not back-solve a `T` to satisfy a `where`.
3. **Untyped lambdas contribute no bound until their input type is fixed.** Inference proceeds in phases: an *implicitly typed* lambda (`x => ...`) is treated as an “un-bound” argument that yields no inference bound until `TSource` has been fixed from some *other* argument. So `static TResult Apply<TSource, TResult>(Func<TSource, TResult> f, TSource s)` infers fine — `s` fixes `TSource`, then the lambda body is bound — but reverse the parameter order or remove `s` entirely and there is

nothing to pin `x`'s type, so inference fails and you must write the type explicitly (`x => x.Length` needs `(string x) => x.Length` or an explicit `Apply<string, int>(...)`).

4. **dynamic and null literals** provide no useful bound. `Identity(null)` cannot infer `T`.

The practical rule: inference flows from *concrete argument types inward*, never from the result outward, and never from constraints. When it fails, supply the type argument explicitly — and note that **C# 14** lets you write `nameof(List<>)` or `nameof(Dictionary<,>.Count)` with *unbound* generic types, which is handy in diagnostics and source-gen attributes where you previously had to invent dummy type arguments.

Recursive generics and CRTP

The Curiously Recurring Template Pattern — `class Foo<TSelf> where TSelf : Foo<TSelf>` — lets a base class refer to its *own most-derived type*, enabling fluent self-returning APIs and `static abstract` factory members that return the concrete type. It is the backbone of `INumber<TSelf>` and `IParsable<TSelf>` in generic math.

```

// CRTP base: TSelf is constrained to be the concrete
// subclass itself.
abstract class Entity<TSelf> where TSelf :
    Entity<TSelf>
{
    public Guid Id { get; private set; } =
        Guid.NewGuid();

    // Returns TSelf, not Entity<TSelf>, so the fluent
    // chain keeps the concrete type.
    public TSelf WithId(Guid id)
    {
        Id = id;
        return (TSelf)this; // safe by construction:
        // the constraint guarantees the shape
    }
}

sealed class Customer : Entity<Customer>
{
    public string Name { get; private set; } = "";
    public Customer WithName(string name) { Name =
        name; return this; }
}

// Fluent chaining preserves the concrete Customer
// type at every step – no casts at the call
// site.
Customer c = new Customer()
    .WithId(Guid.NewGuid()) // returns Customer, not
    Entity<Customer>
    .WithName("Ada");

```

The `(TSelf)this` cast is the one wart: CRTP cannot *force* `Customer : Entity<Customer>` — it only constrains `TSelf` to be *some* `Entity<TSelf>`. A misdeclared `class Evil : Entity<Customer>` (note: `Customer`, not `Evil`) would compile, then throw `InvalidCastException` at runtime when `WithId` executes `(Customer)(Entity<Customer>)evilInstance`. In practice you seal the leaves or accept the constraint as a documented contract; this is the same tradeoff the BCL accepts in `IComparable<T>` self-referential constraints.

Generic specialization tricks: `typeof(T) ==`

`typeof(int)` hot paths

Because each value-type instantiation is JITted separately, a branch like `if (typeof(T) == typeof(int))` is a *compile-time-constant comparison within that instantiation* — the JIT folds it to `true` or `false` and **eliminates the dead branch entirely**. This is the idiomatic way to hand-write a fast path for a specific `T` while keeping a generic fallback. The BCL uses it pervasively (e.g. `Enum`, `Vector<T>`, and serialization fast paths).

```

using System.Numerics;
using System.Runtime.CompilerServices;

static class Summer
{
    public static T Sum<T>(ReadOnlySpan<T> data) where
        T : INumber<T>
    {
        // In the T=int instantiation this whole block
        is JIT-folded and the
        // generic path below is dead-code-eliminated.
        In every other
        // instantiation the int block is removed
        instead.
        if (typeof(T) == typeof(int))
        {
            ReadOnlySpan<int> ints =
                MemoryMarshal.Cast<T, int>(data);
            long acc = 0; //
            widen to avoid overflow in the hot loop
            foreach (int v in ints) acc += v;
            int result = checked((int)acc);
            return Unsafe.As<int, T>(ref result); //
            reinterpret int back to T, no box
        }

        // Generic fallback for every other
        INumber<T>: still allocation-free,
        // still a constrained call to operator +, but
        not the int-specialized loop.
        T total = T.Zero;
        foreach (ref readonly T v in data)
            total += v;
        return total;
    }
}

```

Two details earn their keep. `Unsafe.As<int, T>` (from `System.Runtime.CompilerServices`) reinterprets the `int` result as `T` without boxing — legal here because the surrounding `typeof(T) == typeof(int)` branch proves they are the same type. And the comparison itself costs *nothing at runtime*: it is not a `Type.Equals` call in the generated code; it is a constant the JIT resolved when it specialized the method. The generic fallback path uses the `EqualityComparer`-style constrained-call discipline — `T.Zero` and `operator +` dispatch directly with no boxing.

Contrast this with the naive “default path” that many developers reach for, using `EqualityComparer<T>.Default` to detect a sentinel or comparing values generically: that path is correct and allocation-free, but it cannot vectorize or widen-to-`long`, because it has no knowledge of the concrete type. The `typeof(T)` trick is how you get *both* — a specialized hot loop and a correct generic fallback in one method.

Worked example: a zero-allocation generic statistics pass

This ties the threads together — `unmanaged + INumber<T>` for the math, `allows ref struct` so a stack-only buffer wrapper can flow as a type argument, a ref struct *implementing* an interface (C# 13), a `typeof(T)` specialization with `Unsafe.As` reinterpretation, and constrained calls (`operator +` , `operator <` , `T.CreateChecked`) with no boxing anywhere.

```

using System;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

readonly struct Stats<T>(T sum, T min, T max, int
    count) where T : INumber<T>
{
    public T Sum { get; } = sum;
    public T Min { get; } = min;
    public T Max { get; } = max;
    public int Count { get; } = count;
    public T Mean => Count == 0 ? T.Zero : Sum /
        T.CreateChecked(Count);
}

static class Statistics
{
    // allows ref struct (C# 13) lets a caller pass a
    ref-struct buffer wrapper as TBuffer.
    public static Stats<T> Compute<T, TBuffer>(TBuffer
        buffer)
        where T : unmanaged, INumber<T>
        where TBuffer : IReadOnlySpanProvider<T>,
            allows ref struct
    {
        ReadOnlySpan<T> data = buffer.Span;
        if (data.IsEmpty) return new Stats<T>(T.Zero,
            T.Zero, T.Zero, 0);

        // Specialized fast path: integer sums widen
        to avoid overflow.
        if (typeof(T) == typeof(int))
        {
            ReadOnlySpan<int> ints =
                MemoryMarshal.Cast<T, int>(data);
            long sum = 0; int min = int.MaxValue, max
                = int.MinValue;
            foreach (int v in ints) { sum += v; if (v
                < min) min = v; if (v > max) max = v; }
            return new Stats<T>(
                Reinterpret<int, T>
                (checked((int)sum)),
                Reinterpret<int, T>(min),
                Reinterpret<int, T>(max),
                ints.Length);
        }

        // Generic path – all constrained calls, no
        boxing, no allocation.
        T total = T.Zero, lo = data[0], hi = data[0];

```



```

        foreach (ref readonly T v in data)
        {
            total += v;
            if (v < lo) lo = v;
            if (v > hi) hi = v;
        }
        return new Stats<T>(total, lo, hi,
            data.Length);

        static TOut Reinterpret<TIn, TOut>(TIn value)
            => Unsafe.As<TIn, TOut>(ref value);
    }
}

interface IReadOnlySpanProvider<T> where T : unmanaged
{
    ReadOnlySpan<T> Span { get; } // T in output
                                position only
}

// A ref-struct buffer that can ONLY exist on the
// stack – passed as a type argument
// thanks to `allows ref struct` on TBuffer above.
readonly ref struct StackBuffer<T>(ReadOnlySpan<T>
    span) : IReadOnlySpanProvider<T>
    where T : unmanaged
{
    public ReadOnlySpan<T> Span { get; } = span;
}

// Call site: nothing escapes to the heap.
Span<int> samples = stackalloc int[] { 7, 3, 9, 1, 5
};
var stats = Statistics.Compute<int, StackBuffer<int>>
    (new StackBuffer<int>(samples));
// stats.Sum=25, stats.Min=1, stats.Max=9,
// stats.Count=5, stats.Mean=5

```

Note `IReadOnlySpanProvider<T>` is implemented by a `ref struct` — itself a C# 13 capability (ref structs may implement interfaces, provided the interface is consumed through a generic with `allows ref struct`). The whole pass executes on one JIT-specialized `int` instantiation with the integer fast path live and the generic loop dead-eliminated; for a `double` call you would get the mirror image. No value of `T` is ever boxed, and `StackBuffer<T>` never leaves the stack frame.

Performance and design guidance

- **Reach for `struct` or `unmanaged` constraints when you need devirtualization or zero boxing.** They are the only constraints that guarantee monomorphic, inlinable code, because each value-type instantiation is JITted independently. A bare interface constraint gives you the shared reference-type body with virtual dispatch.
- **Use `EqualityComparer<T>.Default` and `Comparer<T>.Default`, never `T.Equals` / `==` / `CompareTo` directly, on unconstrained `T`.** They are cached, never box, and pick the optimal strategy per type.
- **Use `typeof(T) == typeof(SpecificType)` for hot-path specialization, paired with `Unsafe.As`** to reinterpret without boxing — the JIT folds the comparison to a constant per instantiation, so there is no runtime branch cost.
- **Reserve `allows ref struct` (C# 13) for genuinely span-flowing generic APIs.** It is viral and disables boxing/capture for `T` everywhere downstream; do not add it speculatively.
- **Prefer `IReadOnlyList<out T>` / `IEnumerable<out T>` over arrays when you want covariance.** Array covariance is unsafe, runtime-checked, and costs a type check on every store with an `ArrayTypeMismatchException` waiting at the end.
- **Remember the sharing model when reasoning about optimization.** A trick that depends on the JIT “seeing the concrete type” only fires for value-type `T`; for reference-type `T` you are in the one shared instantiation, and devirtualization there depends on dynamic PGO, not on your constraints.

The unifying idea: constraints are not just compile-time documentation — `struct`, `unmanaged`, and the `allows ref struct` anti-constraint each change *which native code the JIT emits and whether values box*. Choose them for the runtime behavior you want, not merely to satisfy the type checker.

10 · Generic Math and Static Virtual/Abstract Interface Members

Before C# 11, an interface could promise instance behavior but never *static* behavior — so `T.Zero`, `T operator +(T, T)`, or `T.Parse(s)` could not be expressed as a constraint, and “write the algorithm once for every numeric type” was impossible without code generation or `dynamic`. Static abstract interface members closed that gap, and the `System.Numerics` generic-math hierarchy built an entire numeric tower on top of it. This chapter dissects how that dispatch actually works, how to write numeric algorithms that the JIT specializes to branch-free machine code, how to make your own value type a first-class `INumber<T>`, and how C# 14’s user-defined compound assignment operators let those types mutate in place.

Static abstract and static virtual members: what they are and how they dispatch

Introduced in C# 11 (.NET 7), an interface may declare a member as `static abstract` or `static virtual`:

```
public interface IZeroProvider<TSelf> where TSelf :
    IZeroProvider<TSelf>
{
    static abstract TSelf Zero { get; }           //
        every implementer MUST provide
    static virtual TSelf One => TSelf.Zero;       //
        implementer MAY override; default supplied
}
```

The curiously-recurring `where TSelf :`
`IZeroProvider<TSelf>` is the load-bearing idiom. `TSelf` is the implementing type itself, which is what lets a *consumer* write `TSelf.Zero` and have the call resolve to the concrete type’s static member. That is the entire mechanism: **a static abstract member is invoked through a type parameter, never through an instance.**

```

struct Money : IZeroProvider<Money>
{
    public decimal Amount { get; init; }
    public static Money Zero => new() { Amount = 0m };
    // One is inherited via the static virtual default
}

static TSelf SumAll<TSelf>(ReadOnlySpan<TSelf> items)
    where TSelf : IZeroProvider<TSelf>
{
    TSelf total = TSelf.Zero; // dispatched through
                             the type parameter, not an instance
    // ...
    return total;
}

```

Dispatch is *not* virtual in the runtime sense. When `TSelf` is a value type, the JIT instantiates `SumAll<Money>` as a distinct method body with `Money.Zero` resolved to a direct, often-inlined call — there is no v-table lookup, no boxing, and no indirection through an interface dispatch stub. For reference-type instantiations the runtime shares a single generic body and resolves the static member through a per-type dictionary, so there is a small lookup cost but still no boxing. This asymmetry is the whole performance story of generic math: **value-type instantiations monomorphize to straight-line code; reference-type instantiations share code and pay a dictionary lookup**. Compare this with the pre-C#-11 alternatives — an `abstract` instance method on a base class (forces boxing of value types and a v-table call per element) or `dynamic` (a cached DLR call site per operation, orders of magnitude slower than a direct arithmetic instruction). Static abstracts give you the abstraction of an interface with the codegen of a hand-specialized function.

A critical consequence, and a recurring beginner trap: you **cannot** call a static abstract member on an instance. `money.Zero` does not compile. The member lives on the *type*, reachable only as `Money.Zero` or, generically, `TSelf.Zero`. The interface reference `IZeroProvider<Money> x = money; var z = x.Zero;` also fails — there is no such thing as calling a static abstract

through an interface-typed variable, because the variable has erased which concrete type to dispatch to. You always need the type in hand, which in generic code means the type *parameter*.

The generic-math interface hierarchy

`System.Numerics` (.NET 7+) decomposes “being a number” into roughly forty small interfaces, each carrying one capability as static abstract operators. The leaves of interest:

- **Operator interfaces** — `IAdditionOperators<TSelf, TOther, TResult>`, `ISubtractionOperators<>`, `IMultiplyOperators<>`, `IDivisionOperators<>`, `IModulusOperators<>`, `IComparisonOperators<TSelf, TOther, TResult>`, `IEqualityOperators<>`, the shift and bitwise families. Each declares the corresponding `static abstract` operator, e.g. `static abstract TResult operator +(TSelf left, TOther right);`.
- **Identity interfaces** — `IAdditiveIdentity<TSelf, TResult>` (the `AdditiveIdentity` property, i.e. zero) and `IMultiplicativeIdentity<TSelf, TResult>` (one).
- **Parsing/formatting** — `IParsable<TSelf>`, `ISpanParsable<TSelf>`, `ISpanFormattable`, `IUtf8SpanParsable<TSelf>`, `IUtf8SpanFormattable`.

These compose upward into aggregate interfaces.

`INumberBase<TSelf>` pulls in addition, subtraction, multiplication, division, both identities, increment/decrement, unary plus/negation, equality operators, `ISpanParsable`, `ISpanFormattable`, and the all-important `Zero / One / Abs / IsNaN / CreateChecked / CreateSaturating / CreateTruncating` static abstracts. `INumber<TSelf>` extends `INumberBase<TSelf>` and adds `IComparisonOperators<TSelf, TSelf, bool>`, `IModulusOperators<TSelf, TSelf, TSelf>`, `IComparable`, and helpers like `Clamp`, `Max`, `Min`, `Sign`. Above those sit `IBinaryInteger<TSelf>`, `IFloatingPoint<TSelf>`, `IBinaryFloatingPointIeee754<TSelf>`, `ISignedNumber<TSelf>`, `IUnsignedNumber<TSelf>`.

Every built-in numeric — `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `Int128`, `UInt128`, `Half`, `float`, `double`, `decimal`, `System.Numerics.BigInteger`, `Complex` (a subset) — implements the appropriate slice. The practical upshot: **constrain `T` to the smallest interface your algorithm actually needs**. If you only add, require `IAdditionOperators<T, T, T>` plus `IAdditiveIdentity<T, T>`; that admits more types (including non-`INumber` user types) and documents intent. If you need ordering, comparison, and a literal zero, `INumber<T>` is the convenient catch-all.

Writing a numeric algorithm once: Sum

The canonical first example. We want a `Sum` that works for `int`, `double`, `decimal`, `Fixed16`, and anything else numeric, with zero boxing and full JIT specialization:

```
using System.Numerics;

public static class Reductions
{
    public static T Sum<T>(ReadOnlySpan<T> values)
        where T : INumber<T>
    {
        T total = T.Zero; // static
                          // abstract: the type's additive identity
        foreach (T v in values)
            total += v; // resolves
                       // to T.op_Addition(total, v)
        return total;
    }
}

// Usage — each call site instantiates a specialized
// method body:
int si = Reductions.Sum<int>([1, 2, 3, 4]);
// 10
double sd = Reductions.Sum([1.5, 2.5, 3.0]);
// 7 (T inferred as double)
decimal sm = Reductions.Sum<decimal>([0.1m, 0.2m]);
// 0.3 (exact, decimal)
```

`T.Zero` is `IAdditiveIdentity` surfaced through `INumberBase`; `total += v` lowers to the static operator `+`.

For `Sum<int>` the JIT emits an `add` instruction in a tight loop — identical to a hand-written `int` sum. For `Sum<double>` it emits `addsd`. No allocation, no virtual dispatch. The collection-expression arguments (`[1, 2, 3, 4]`) are a C# 12 (.NET 8) feature; here they target `ReadOnlySpan<T>`, which the compiler can stack-allocate, so the whole call is allocation-free.

Average needs a division and a count. The subtlety is converting the integer count to `T`:

```
public static T Average<T>(ReadOnlySpan<T> values)
    where T : INumber<T>
{
    if (values.IsEmpty) throw new
        ArgumentException("empty", nameof(values));
    T sum = Sum(values);
    T count = T.CreateChecked(values.Length); // int
        -> T, throwing on overflow
    return sum / count;
}
```

`T.CreateChecked(values.Length)` is the bridge from the `int` length to the generic `T`. We return to `Create*` semantics in detail below. A hand-written `Clamp` shows constraint minimalism — it only needs ordering, which `INumber<T>` carries through `IComparisonOperators<T, T, bool>`, so the relational operators on `T` are directly usable:

```
public static T Clamp<T>(T value, T min, T max) where
    T : INumber<T>
{
    if (min > max) throw new ArgumentException("min >
        max");
    return value < min ? min : value > max ? max :
        value; // IComparisonOperators
}
```

`<` and `>` here resolve to `IComparisonOperators<T, T, bool>.op_LessThan` etc. (Note `INumber<T>` already ships a static `Clamp` — this is illustrative.)

Create: CreateChecked, CreateSaturating, CreateTruncating

`INumberBase<TSelf>` declares three static abstract conversion factories, each taking a `TOther : INumberBase<TOther>` and producing a `TSelf`. They differ only in out-of-range behavior:

Method	Out-of-range behavior
<code>CreateChecked<TOther>(value)</code>	Throws <code>OverflowException</code>
<code>CreateSaturating<TOther>(value)</code>	Clamps to <code>TSelf.MinValue</code> / <code>TSelf.MaxValue</code>
<code>CreateTruncating<TOther>(value)</code>	Wraps (keeps the low bits / truncates)

```
byte a = byte.CreateChecked(200);           // 200
// byte b = byte.CreateChecked(384);         // throws
//                                     OverflowException
byte c = byte.CreateSaturating(384);         // 255
//                                     (clamped to byte.MaxValue)
byte d = byte.CreateTruncating(384);         // 128 (384
//                                     & 0xFF == 0x80)
byte e = byte.CreateTruncating(-1);          // 255
//                                     (wraps)

// Cross-width, signed/unsigned, and float->int all go
// through the same APIs:
int fi = int.CreateSaturating(double.PositiveInfinity);
// int.MaxValue
short sh = short.CreateTruncating(70000);
// 4464 (low 16 bits: 70000 & 0xFFFF == 0x1170)
short sg = short.CreateTruncating(100000);
// -31072 (low 16 bits 0x86A0; bit 15 set => negative)
```

For IEEE-754 destinations (`float`, `double`, `Half`) all three behave as `CreateSaturating`, mapping out-of-range to `±Infinity` and preserving `NaN`, because those types have no “overflow” — they have infinities. This is a frequent surprise: `double.CreateChecked(BigInteger huge)` does not throw; it returns `+Infinity`.

In a generic method you reach these through the type parameter:

```
// Safely re-bucket a span of one numeric type into  
another, generic over both ends.  
public static TOut[] ConvertChecked<TIn, TOut>  
    (ReadOnlySpan<TIn> src)  
    where TIn : INumberBase<TIn>  
    where TOut : INumberBase<TOut>  
{  
    var dst = new TOut[src.Length];  
    for (int i = 0; i < src.Length; i++)  
        dst[i] = TOut.CreateChecked(src[i]); //  
        throws if any element won't fit TOut  
    return dst;  
}  
  
double[] price = [19.99, 9.50, 4.25];  
// Throws on the first non-integer or out-of-range  
element – fail loud:  
// int[] cents = ConvertChecked<double, int>(price);  
// OK: 19, 9, 4 (truncates fraction!)
```

A subtle correctness note in that last line: `CreateChecked` checks *range*, not *fractional loss*. `int.CreateChecked(19.99)` returns `19` without complaint — the value is in `int`'s range, the fraction is silently dropped. “Checked” means “checked for overflow,” not “checked for exactness.”

Pitfall — mixing `CreateChecked` vs `CreateTruncating` and getting silent wraparound vs an exception. This is the single most common generic-math bug. Picture a generic histogram that downcasts accumulated `long` counts into a `T` bucket. With `CreateTruncating` an overflowing count silently wraps to a small or negative number and your statistics are quietly wrong; with `CreateChecked` you get a loud `OverflowException` at the exact offending element. Choose deliberately: `CreateTruncating` only when you *want* modular/bit-level semantics (hashing, fixed-width arithmetic, deliberate wrap), `CreateSaturating` for clamp-to-range UI/DSP scenarios, and `CreateChecked` as the safe default for “this should always fit, tell me if it doesn’t.” Reaching for `CreateTruncating` because it “never throws” is how you ship a wraparound bug.

Generic parsing with IParsable and ISpanParsable

Before C# 11 you could not write one generic `Parse<T>` ; you needed an overload per type or a `switch` on `typeof` .
`IParsable<TSelf>` (string) and `ISpanParsable<TSelf>` (`ReadOnlySpan<char>` , allocation-free) fix that with static abstract `Parse` / `TryParse` :

```
using System.Globalization;

public static class GenericParse
{
    // One method replaces int.Parse / double.Parse /
    // decimal.Parse / ...
    public static T Parse<T>(ReadOnlySpan<char> s,
        IFormatProvider? provider = null)
        where T : ISpanParsable<T>
        => T.Parse(s, provider ??
            CultureInfo.InvariantCulture);

    public static bool TryParse<T>(ReadOnlySpan<char>
        s, out T value)
        where T : ISpanParsable<T>
        => T.TryParse(s, CultureInfo.InvariantCulture,
            out value);
}

int i = GenericParse.Parse<int>("42");
// 42
double d = GenericParse.Parse<double>("3.14");
// 3.14
decimal m = GenericParse.Parse<decimal>("9.99");
// 9.99
bool ok = GenericParse.TryParse<long>("not a number",
    out long _); // false
```

Because `ISpanParsable<T>` takes a `ReadOnlySpan<char>` , parsing a substring is allocation-free — no `Substring` call. A common generic CSV/column reader can parse a row of mixed column types by calling `Parse<TColumn>` with the right `T` for each field, all monomorphized. Prefer `ISpanParsable<T>` over `IParsable<T>` in new code; the string overload is trivially expressible by passing `s.AsSpan()` , and the span form avoids forcing callers to allocate strings.

Implementing INumber: a fixed-point Fixed16

To make a user type first-class in generic math, implement the relevant interfaces and provide every static abstract member. Here is a Q16.16 signed fixed-point type (32-bit, 16 integer + 16 fractional bits) implementing enough of `INumber<Fixed16>` to be usable. The point is to show the *shape* of a real implementation — including the checked operators and the `Create*` plumbing — not to be production-complete.

```

using System.Globalization;
using System.Numerics;

public readonly struct Fixed16 :
    INumber<Fixed16>,
    ISignedNumber<Fixed16>
{
    private const int  FracBits = 16;
    private const long Scale     = 1L << FracBits; //
        65536
    private readonly int _raw; //
        value * 65536, two's complement

    private Fixed16(int raw) => _raw = raw;
    public static Fixed16 FromRaw(int raw) =>
        new(raw);

    public static Fixed16 FromDouble(double d) =>
        new(checked((int)Math.Round(d * Scale)));
    public double ToDouble() => (double)_raw / Scale;

    // ---- Identities & well-known values (static
        abstract on INumberBase) ----
    public static Fixed16 Zero          => new(0);
    public static Fixed16 One           =>
        new((int)Scale);
    public static Fixed16 NegativeOne   =>
        new((int)-Scale);
    public static Fixed16 AdditiveIdentity =>
        Zero;
    public static Fixed16 MultiplicativeIdentity =>
        One;
    public static int Radix => 2;

    // ---- Arithmetic operators (unchecked + paired
        checked variants) ----
    public static Fixed16 operator +(Fixed16 a,
        Fixed16 b) => new(a._raw + b._raw);
    public static Fixed16 operator checked +(Fixed16
        a, Fixed16 b) => new(checked(a._raw +
        b._raw));
    public static Fixed16 operator -(Fixed16 a,
        Fixed16 b) => new(a._raw - b._raw);
    public static Fixed16 operator checked -(Fixed16
        a, Fixed16 b) => new(checked(a._raw -
        b._raw));
    public static Fixed16 operator -(Fixed16 v) =>
        new(-v._raw);
    public static Fixed16 operator +(Fixed16 v) => v;

    // Multiply: widen to 64-bit so the intermediate
        product doesn't overflow, then rescale.

```

```

public static Fixed16 operator *(Fixed16 a,
    Fixed16 b)
    => new((int)((long)a._raw * b._raw >>
        FracBits));
public static Fixed16 operator checked *(Fixed16
    a, Fixed16 b)
    => new(checked((int)((long)a._raw * b._raw >>
        FracBits)));
public static Fixed16 operator /(Fixed16 a,
    Fixed16 b)
    => new((int)((long)a._raw << FracBits) /
        b._raw));
public static Fixed16 operator %(Fixed16 a,
    Fixed16 b) => new(a._raw % b._raw);

public static Fixed16 operator ++(Fixed16 v) =>
    new(v._raw + (int)Scale);
public static Fixed16 operator --(Fixed16 v) =>
    new(v._raw - (int)Scale);

// ---- Equality & comparison operators (static
//      abstract) ----
public static bool operator ==(Fixed16 a, Fixed16
    b) => a._raw == b._raw;
public static bool operator !=(Fixed16 a, Fixed16
    b) => a._raw != b._raw;
public static bool operator < (Fixed16 a, Fixed16
    b) => a._raw < b._raw;
public static bool operator > (Fixed16 a, Fixed16
    b) => a._raw > b._raw;
public static bool operator <=(Fixed16 a, Fixed16
    b) => a._raw <= b._raw;
public static bool operator >=(Fixed16 a, Fixed16
    b) => a._raw >= b._raw;

public int CompareTo(Fixed16 other) =>
    _raw.CompareTo(other._raw);
public int CompareTo(object? obj) => obj is
    Fixed16 f ? CompareTo(f) : 1;
public bool Equals(Fixed16 other) => _raw ==
    other._raw;
public override bool Equals(object? obj) => obj is
    Fixed16 f && Equals(f);
public override int GetHashCode() => _raw;

// ---- INumber / INumberBase static helpers ----
public static Fixed16 Abs(Fixed16 v) =>
    new(Math.Abs(v._raw));
public static bool IsZero(Fixed16 v) => v._raw ==
    0;
public static bool IsNegative(Fixed16 v) => v._raw
    < 0;
public static bool IsPositive(Fixed16 v) => v._raw
    > 0;

```

```

public static bool IsInteger(Fixed16 v) => (v._raw
    & (Scale - 1)) == 0;
public static bool IsEvenInteger(Fixed16 v) =>
    IsInteger(v) && ((v._raw >> FracBits) & 1) ==
    0;
public static bool IsOddInteger(Fixed16 v) =>
    IsInteger(v) && ((v._raw >> FracBits) & 1) ==
    1;
public static bool IsCanonical(Fixed16 v) => true;
public static bool IsComplexNumber(Fixed16 v) =>
    false;
public static bool IsRealNumber(Fixed16 v) =>
    true;
public static bool IsImaginaryNumber(Fixed16 v) =>
    false;
public static bool IsFinite(Fixed16 v) => true;
public static bool IsInfinity(Fixed16 v) => false;
public static bool IsPositiveInfinity(Fixed16 v)
    => false;
public static bool IsNegativeInfinity(Fixed16 v)
    => false;
public static bool IsNaN(Fixed16 v) => false;
public static bool IsNormal(Fixed16 v) => v._raw
    != 0;
public static bool IsSubnormal(Fixed16 v) =>
    false;
public static int Sign(Fixed16 v) =>
    Math.Sign(v._raw);

public static Fixed16 MaxMagnitude(Fixed16 x,
    Fixed16 y) => Abs(x) >= Abs(y) ? x : y;
public static Fixed16 MaxMagnitudeNumber(Fixed16
    x, Fixed16 y) => MaxMagnitude(x, y);
public static Fixed16 MinMagnitude(Fixed16 x,
    Fixed16 y) => Abs(x) <= Abs(y) ? x : y;
public static Fixed16 MinMagnitudeNumber(Fixed16
    x, Fixed16 y) => MinMagnitude(x, y);

// ---- Conversions: the Create* triad routes
// through TryConvert* hooks ----
public static Fixed16 CreateChecked<TOther>(TOther
    value) where TOther : INumberBase<TOther>
    => FromDouble(double.CreateChecked(value));
// simplistic but valid
public static Fixed16 CreateSaturating<TOther>
    (TOther value) where TOther :
    INumberBase<TOther>
{
    double d = double.CreateSaturating(value) *
    Scale;
    int raw = d >= int.MaxValue ? int.MaxValue : d
    <= int.MinValue ? int.MinValue : (int)d;
    return new(raw);
}

```

```

public static Fixed16 CreateTruncating<TOther>
    (TOther value) where TOther :
    INumberBase<TOther>
    => new<unchecked>((int)
        (long.CreateTruncating(value) << FracBits));

static bool
    INumberBase<Fixed16>.TryConvertFromChecked<T>
    (T v, out Fixed16 r) { r =
        CreateChecked(v); return true; }
static bool
    INumberBase<Fixed16>.TryConvertFromSaturating<
    T>(T v, out Fixed16 r) { r =
        CreateSaturating(v); return true; }
static bool
    INumberBase<Fixed16>.TryConvertFromTruncating<
    T>(T v, out Fixed16 r) { r =
        CreateTruncating(v); return true; }
static bool
    INumberBase<Fixed16>.TryConvertToChecked<T>
    (Fixed16 v, out T r) { r =
        T.CreateChecked(v.ToDouble()); return true;
    }
static bool
    INumberBase<Fixed16>.TryConvertToSaturating<T>
    (Fixed16 v, out T r) { r =
        T.CreateSaturating(v.ToDouble()); return true;
    }
static bool
    INumberBase<Fixed16>.TryConvertToTruncating<T>
    (Fixed16 v, out T r) { r =
        T.CreateTruncating(v.ToDouble()); return true;
    }

// ---- Parsing & formatting ----
public static Fixed16 Parse(ReadOnlySpan<char> s,
    IFormatProvider? p) =>
    FromDouble(double.Parse(s, p));
public static Fixed16 Parse(string s,
    IFormatProvider? p) =>
    FromDouble(double.Parse(s, p));
public static Fixed16 Parse(ReadOnlySpan<char> s,
    NumberStyles st, IFormatProvider? p) =>
    FromDouble(double.Parse(s, st, p));
public static Fixed16 Parse(string s, NumberStyles
    st, IFormatProvider? p) =>
    FromDouble(double.Parse(s, st, p));
public static bool TryParse(ReadOnlySpan<char> s,
    IFormatProvider? p, out Fixed16 r)
    => Try(double.TryParse(s, p, out double d), d,
    out r);
public static bool TryParse(string? s,
    IFormatProvider? p, out Fixed16 r)
    => Try(double.TryParse(s, p, out double d), d,
    out r);

```

```

public static bool TryParse(ReadOnlySpan<char> s,
    NumberStyles st, IFormatProvider? p, out
    Fixed16 r)
    => Try(double.TryParse(s, st, p, out double
    d), d, out r);
public static bool TryParse(string? s,
    NumberStyles st, IFormatProvider? p, out
    Fixed16 r)
    => Try(double.TryParse(s, st, p, out double
    d), d, out r);
private static bool Try(bool ok, double d, out
    Fixed16 r) { r = ok ? FromDouble(d) : default;
    return ok; }

public string ToString(string? fmt,
    IFormatProvider? p) =>
    ToDouble().ToString(fmt, p);
public override string ToString() =>
    ToDouble().ToString(CultureInfo.InvariantCulture);
public bool TryFormat(Span<char> dest, out int
    written, ReadOnlySpan<char> fmt,
    IFormatProvider? p)
    => ToDouble().TryFormat(dest, out written,
    fmt, p);
}

```

Now `Fixed16` flows through every generic-math algorithm written earlier with no further work:

```

ReadOnlySpan<Fixed16> grid = [Fixed16.FromDouble(1.5),
    Fixed16.FromDouble(2.25), Fixed16.One];
Fixed16 total = Reductions.Sum(grid); // 4.75,
    exact in Q16.16
Fixed16 mean = Reductions.Average(grid); //
    1.5833... (uses CreateChecked internally)
Fixed16 p = GenericParse.Parse<Fixed16>("3.5");
    // ISpanParsable route

```

Pitfall — implementing `INumber<T>` partially and discovering required members at the use site. `INumber<T>`

transitively requires *dozens* of static abstract members. If you stub out only the operators you “think you need,” the type will not compile as an `INumber<Fixed16>` until every member — including obscure ones like `IsCanonical`, `MaxMagnitudeNumber`, `Radix`, and all six `TryConvert*` hooks — is present. The compiler reports each missing member as a separate “does not implement interface member” error, and the list is long. Two cop-

ing strategies: (1) constrain to the *smallest* interface you need (`IAdditionOperators` + `IAdditiveIdentity`) rather than the whole `INumber<T>` if your callers only add; (2) when you do want the full thing, copy the member list from an existing reference implementation and fill in bodies, because IntelliSense’s “implement interface” quick-fix will scaffold all the signatures for you. Note the explicit-interface-implementation syntax on `TryConvertFromChecked` etc.: those are declared on `INumberBase<TSelf>` as static abstract and are conventionally implemented explicitly so they don’t pollute the public surface.

User-defined compound assignment operators (C# 14) and in-place mutation

For small value types, `total += v` lowering to `total = T.op_Addition(total, v)` is fine — the copy is a register move. For *large* types (a `BigInteger`, a tensor, an arbitrary-precision accumulator) the compound operator allocates a brand-new instance, copies the entire payload, and discards the old one. C# 14 (.NET 10) adds **user-defined compound assignment operators** so the type can mutate the existing instance in place instead.

The syntax, per the language spec, is deliberately different from a binary operator: the compound operator is an **instance** member (not `static`), has `void` return type, and takes the right-hand operand as its single parameter. Increment/decrement get an instance form too — `public void operator ++()` taking no parameters:

```

using System.Numerics;

// A growable big-integer accumulator whose += mutates
// in place instead of reallocating.
public sealed class BigAccumulator
{
    private BigInteger _value;
    public BigAccumulator(BigInteger seed = default)
        => _value = seed;
    public BigInteger Value => _value;

    // C# 14: instance, void-returning compound
    // assignment. Mutates _value in place.
    public void operator +=(BigInteger rhs) => _value
        += rhs;
    public void operator -=(BigInteger rhs) => _value
        -= rhs;
    public void operator *=(BigInteger rhs) => _value
        *= rhs;

    // C# 14 instance increment: no parameter, void
    // return.
    public void operator ++() => _value +=
        BigInteger.One;

    // A checked variant must be paired with the
    // regular one (spec rule).
    public void operator checked +=(BigInteger rhs) =>
        _value = checked(_value + rhs);

    public override string ToString() =>
        _value.ToString();
}

```

Usage and what the compiler emits:

```

var acc = new BigAccumulator(BigInteger.Pow(10, 40));
for (int i = 0; i < 1_000_000; i++)
    acc += i;           // lowers to
                        // acc.op_AdditionAssignment(i) - NO new
                        // BigAccumulator allocated
acc++;                 // lowers to
                        // acc.op_IncrementAssignment()
Console.WriteLine(acc.Value);

```

Because `BigAccumulator` is a reference type, the runtime evaluates `acc` once to an instance and invokes the instance operator on it; the variable `acc` is never reassigned. Contrast the pre-

C#-14 world: without an instance `+=`, you would have needed a `static BigAccumulator operator +(BigAccumulator, BigInteger)` returning a *new* accumulator, allocating one object per loop iteration — a million allocations and a million GC-tracked objects here. The instance operator turns that into a single object mutated a million times — the allocation count for the accumulation loop drops from one-per-iteration to effectively zero, which is the whole motivation the language design notes give for the feature (large or heap-backed payloads like `BigInteger` and tensors).

Three rules to internalize. **First**, the operator is `void` and instance — if you write `public static BigAccumulator operator +=` it will not compile; that's not the new feature, that's the old “synthesize from `+`” path which still requires only `operator +`. **Second**, a `checked` compound operator (`operator checked +=`) must be paired with a regular `operator +=` declaration or you get a compile error — same pairwise rule as C# 11's checked binary operators. **Third**, you *may* mark the operator `readonly` in a struct (the spec allows it) but it is almost always pointless: the whole reason to define a compound operator is to mutate the instance, and `readonly` forbids that. Reach for these operators only when in-place mutation is a measurable win — i.e. the type owns a large or heap-backed payload. For a small immutable value type, defining them buys nothing over the synthesized path and adds surface area.

There is also an interaction worth flagging: the result of a compound assignment is the *post*-mutation value. For a reference type the compiler simply mutates the instance in place and yields that same instance, so even `var x = acc += y;` allocates nothing extra. For a value type the picture is different — when the result is *used* (`var x = s += y;`), the spec requires the receiver to be evaluated to a temporary, mutated, written back, and the temporary returned, so a copy is unavoidable; when the result is *discarded* (`s += y;`), the operator is invoked directly on `s` with no copy. The optimization is therefore fullest for val-

ue types when the result is discarded — the common loop-accumulator case — which is exactly where it matters.

A static virtual member for a self-describing Identity pattern

`static virtual` (not `abstract`) lets an interface ship a *default* static implementation that implementers may override. This models a “self-describing identity” cleanly: a type declares its neutral element via a `static abstract Zero`, and the interface supplies a default `Identity` *derived from that zero* through a `static virtual` member — which an implementer may override (a matrix type would override it to the identity matrix). Keeping the interface self-contained (it declares its own `Zero` rather than constraining `TSelf` to `INumberBase<TSelf>`) means a type can opt in cheaply, without taking on the whole numeric tower:

```

public interface IIdentity<TSelf> where TSelf :
    IIdentity<TSelf>
{
    static abstract TSelf Zero { get; } //
        must be provided
    static virtual TSelf Identity => TSelf.Zero; //
        default = additive identity; may be overridden
    static abstract string IdentityName { get; } //
        must be provided
}

public readonly record struct Modular7(int Value) :
    IIdentity<Modular7>
{
    public static Modular7 operator +(Modular7 a,
        Modular7 b) => new((a.Value + b.Value) % 7);
    public static Modular7 Zero => new(0);
    // Inherit the static virtual default Identity =>
    // Zero; only the human-readable name is bespoke.
    public static string IdentityName => "0 (mod 7)";
}

// Generic consumer reaches the static members through
// the type parameter:
static string Describe<T>() where T : IIdentity<T>
    => $"typeof(T).Name: identity is {T.Identity},
        named '{T.IdentityName}'";

// Console.WriteLine(Describe<Modular7>());
// Modular7: identity is Modular7 { Value = 0 }, named
// '0 (mod 7)'

```

`T.Identity` dispatches to the override if present, else the static virtual default — here `Modular7` supplies no override, so

`T.Identity` resolves to the interface default `=> TSelf.Zero`, which in turn dispatches to `Modular7.Zero`. All of this is resolved at JIT time per instantiation, with no v-table.

`T.IdentityName` and `T.Zero` are static abstract, so every implementer must supply them. Again: these are reachable only via the type parameter `T`, never via an instance of `T`.

Generic math meets SIMD and TensorPrimitives

The generic-math abstractions and the *vectorized* fast paths meet in two places. `System.Numerics.Vector<T>` is constrained to the primitive numeric element types and lets you

write SIMD loops generically over those, but it does **not** accept arbitrary `INumber<T>` user types — SIMD needs a fixed register layout. So `Vector<Fixed16>` is not a thing;

`Vector<int>` / `Vector<float>` are.

The higher-level entry point is

`System.Numerics.Tensors.TensorPrimitives` (the `System.Numerics.Tensors` NuGet package, with a netstandard2.0 baseline and the broadly-generic surface added in .NET 9), which provides hardware-accelerated `Sum`, `Add`, `Multiply`, `Dot`, `Cosine`, `Norm`, etc. over `ReadOnlySpan<T>`. The .NET 9 release expanded it from roughly 40 non-generic overloads to nearly 200, most of them generic. Each method constrains `T` to the *smallest* generic-math interface it needs — `Add` is where `T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>`, `Max` is where `T : INumber<T>`, the trigonometric and norm methods to floating-point interfaces — and internally dispatches to AVX-512/AVX2/NEON kernels for the primitive types while falling back to a scalar generic loop for everything else:

```
using System.Numerics.Tensors;

float[] a = [1, 2, 3, 4, 5, 6, 7, 8];
float[] b = [8, 7, 6, 5, 4, 3, 2, 1];

float dot = TensorPrimitives.Dot<float>(a, b);    // 120, SIMD-accelerated
float sum = TensorPrimitives.Sum<float>(a);        // 36

var dst = new float[a.Length];
TensorPrimitives.Add<float>(a, b, dst);            // elementwise, vectorized
```

The design lesson: **your generic-math algorithm is correct for all `T`, but only fast for the element types the run-time has vector kernels for.** For a hot reduction over

`float` / `double` / `int`, call `TensorPrimitives` rather than your hand-rolled `Sum<T>` — it will be several times faster on a wide vector unit. For a user type like `Fixed16`, `TensorPrimitives` (where generic) falls back to the scalar path, identical in speed to your own loop. Don't try to force SIMD on a

non-primitive `INumber<T>` ; it isn't supported, and the scalar generic path is the intended behavior.

Worked example: a generic, allocation-aware running statistics accumulator

This ties the chapter together — static abstracts for the numeric constraint, `Create*` for the safe count conversion, generic parsing to ingest text, and a C# 14 in-place compound operator on the large-payload accumulator:

```

using System.Numerics;

// Welford-free simple running stats, generic over any
// INumber, with an in-place += for streaming.
public sealed class RunningStats<T> where T :
    INumber<T>
{
    private T _sum = T.Zero;
    private T _min;
    private T _max;
    private int _count;

    // C# 14 instance compound operator: fold a new
    // sample in WITHOUT building a new accumulator.
    public void operator +=(T sample) => Add(sample);

    // Same fold, callable as an ordinary method.
    // `this += sample` would NOT compile here:
    // for a class, `this` is a value (not a
    // variable), so the compound-assignment operator
    // path doesn't kick in – call the operator on an
    // external variable, or a method like this.
    private void Add(T sample)
    {
        if (_count == 0) { _min = sample; _max =
            sample; }
        else { if (sample < _min) _min = sample; if
            (sample > _max) _max = sample; }
        _sum += sample; // T.op_Addition on
            the element
        _count++;
    }

    public T Sum => _sum;
    public T Min => _min;
    public T Max => _max;
    public int Count => _count;

    public T Mean => _count == 0
        ? throw new InvalidOperationException("no
            samples")
        : _sum / T.CreateChecked(_count); // int
            count -> T, throwing if it somehow won't fit

    // Ingest a line of whitespace-separated numbers
    // via ISpanParsable<T>, zero substring allocs.
    public void AddLine(ReadOnlySpan<char> line)
    {
        foreach (Range r in line.Split(' '))
        {
            ReadOnlySpan<char> tok = line[r].Trim();

```



```

        if (!tok.IsEmpty && T.TryParse(tok,
            provider: null, out T value))
            Add(value); // folds
            in place – no allocation per sample
        }
    }

// --- driver ---
var stats = new RunningStats<double>();
stats.AddLine("1.5 2.5 3.0 0.5 4.0");
stats += 10.0; // direct
            in-place fold

Console.WriteLine($"n={stats.Count} sum={stats.Sum}
    min={stats.Min} max={stats.Max} mean=
    {stats.Mean}");
// n=6 sum=21.5 min=0.5 max=10 mean=3.5833333333333335

// Works unchanged for decimal (exact money math) or
// Fixed16 (fixed-point DSP):
var money = new RunningStats<decimal>();
foreach (decimal d in (decimal[])[19.99m, 9.50m,
    4.25m]) money += d;
Console.WriteLine($"avg price = {money.Mean:C}"); //
    avg price = $11.25

```

Every numeric operation here is monomorphized:

`RunningStats<double>` compiles to `double` arithmetic, `RunningStats<decimal>` to `decimal` arithmetic, with no boxing and no `dynamic`. The allocation-free `ReadOnlySpan<char>.Split` overload (introduced in .NET 9, returning a `SpanSplitEnumerator<char>` whose `Current` is a `Range` into the original span) plus `T.TryParse(span, ...)` parses each token without allocating a `string`. The external `stats += 10.0` call site invokes the instance compound operator (`op_AdditionAssignment`), so streaming samples mutates one `RunningStats<T>` object rather than allocating a fresh one per fold. Inside `AddLine` we call the shared `Add` helper rather than `this += value`, because in a class method `this` is classified as a *value*, not a variable, so the C# 14 compound-assignment operator path does not apply to it — a subtlety worth internalizing. And `T.CreateChecked(_count)` is the only safe way to turn the `int` `count` into `T` for the mean — `(T)_count` does not

compile, because there is no unconstrained cast from `int` to an open `T`.

When not to reach for generic math

Generic math is the right tool when an algorithm is genuinely numeric-type-agnostic and you have more than one element type to serve. It is the *wrong* tool when you have exactly one element type (`int`), in which case a plain non-generic method is simpler, compiles faster, and produces identical code. It is also a poor fit when “number-like” behavior is only superficially similar across your types — forcing a `Money`, a `Percentage`, and a `Temperature` under one `INumber<T>` constraint invites unit-mixing bugs that the type system would otherwise have caught (adding a temperature to a price compiles fine under `INumber<T>`). And remember the codegen cost model: every distinct value-type instantiation produces a separate JIT-compiled method body, so a generic-math method instantiated over fifteen numeric types yields fifteen code bodies — usually a non-issue, but worth knowing for AOT size budgets. Constrain to the narrowest interface, prefer `TensorPrimitives` for hot primitive-element reductions, reach for C# 14 instance compound operators only when the payload is large enough that the saved allocation pays for the extra API surface, and always pick the `Create*` variant whose overflow behavior matches your actual correctness requirement rather than the one that happens not to throw.

11 · Delegates, Lambdas, and Expression Trees

Delegates are the oldest first-class function abstraction in C#, and they sit underneath almost everything modern: LINQ, async continuations, event handlers, dependency-injection factories, and the predicate-driven APIs of the BCL. This chapter takes you down to the metal — invocation lists, display classes, method-group caching, and the IL the compiler actually emits — then back up to expression trees, where code becomes inspectable data you can rewrite and recompile at runtime. The goal is judgment: knowing which of these tools costs an allocation, which costs a JIT, and which costs nothing at all.

Delegate internals: invocation lists and multicast

A delegate is not a function pointer. It is a sealed reference type derived from `System.MulticastDelegate`, carrying two essential fields: a `Target` (the receiver `this`, or `null` for static methods) and a method pointer. The “multicast” part means every delegate is potentially a *list* of `(target, method)` pairs — the invocation list. Single-target delegates store that pair inline; combining delegates with `+` materializes an internal array.

```
Action a = () => Console.Write("A");
Action b = () => Console.Write("B");

Action chain = a + b;           // multicast:
    invocation list = [a, b]
chain += () => Console.Write("C");

chain();                        // prints ABC
Console.WriteLine();
Console.WriteLine(chain.GetInvocationList().Length);
// 3
```

Three semantic rules trip up even experienced developers:

1. **Delegates are immutable.** `chain += x` does not mutate `chain`; it allocates a new delegate whose invocation list is

the concatenation, then rebinds the local. This is why `+=` on an `event` is thread-safe at the language level (the compiler emits an interlocked compare-exchange loop) but `+=` on a plain field is not.

- 2. Multicast invocation returns only the last result.** For a `Func<int>` chain of three methods, the runtime walks the list in order and discards every return value except the final one. If you need all results, walk `GetInvocationList()` yourself.
- 3. An exception aborts the walk.** If the second of three subscribers throws, the third never runs and the exception propagates to the caller. Robust event raisers iterate the invocation list manually and aggregate failures:

```
static void RaiseSafely(Action? handlers)
{
    if (handlers is null) return;
    List<Exception>? errors = null;
    foreach (Action handler in
        handlers.GetInvocationList()) // each element
                                     // is single-cast
    {
        try { handler(); }
        catch (Exception ex) { (errors ??=
            []).Add(ex); } // C# 12 collection
                           // expression
    }
    if (errors is { Count: > 0 }) throw new
        AggregateException(errors);
}
```

Note the loop-variable type: `GetInvocationList()` returns `Delegate[]`, and `foreach` is special-cased to apply an *explicit* (narrowing) conversion from the array's element type to the loop variable's type — so `foreach (Action handler in ...)` emits a per-element `castclass Action` for you. It is not covariance and not free; if a non-`Action` delegate ever sneaked into the array it would throw `InvalidCastException` at iteration. Each returned element is itself single-cast, so calling it invokes exactly one target.

Action, Func, and Predicate

You almost never declare your own delegate types anymore. The BCL ships `Action<...>` (up to 16 args, returns `void`), `Func<..., TResult>` (up to 16 args plus a result), and `Predicate<T>` (a `Func<T, bool>` that predates generics-heavy LINQ and survives mostly in `List<T>.FindAll`, `Array.FindIndex`, etc.). `Predicate<T>` and `Func<T, bool>` are *distinct, non-convertible types* despite identical shapes — delegate identity is nominal, not structural, so you cannot pass a `Func<int, bool>` where a `Predicate<int>` is expected without an explicit wrap. Prefer `Func` / `Action` in new APIs; reserve named delegates for cases where a descriptive name aids readability or where you need `ref` / `out` / `in` parameters that the generic family cannot express.

Lambda natural types, target typing, and inferred returns

Before C# 10 a lambda had no type of its own; it only acquired one when assigned to a delegate or `Expression`. Since C# 10 a lambda (and a method group) has a **natural type** when the compiler can infer one, which is what makes `var` work:

```
var parse = (string s) => int.Parse(s); // natural
                                     type: Func<string, int>
var greet = () => Console.WriteLine("hi"); // natural
                                     type: Action
var add    = (int x, int y) => x + y;      //
                                     Func<int, int, int>
```

Natural typing has limits worth internalizing. The compiler synthesizes a `Func<>` / `Action<>` only when every parameter type is explicit (or inferable) and the return type is consistent. A lambda body with divergent branch types (`true ? 1 : "x"`) has no natural return type and won't bind to `var`. **Target typing** is the inverse and still applies: when a delegate or expression-tree type is supplied by context, the lambda adapts to it, which lets the same syntax become a `Func`, a `Predicate`, or an `Expression<Func<...>>`:

```
Func<int, bool> isEven      = n => n % 2 == 0;
    // delegate target
Expression<Func<int, bool>> e = n => n % 2 == 0;
    // expression-tree target - compiled to data,
    not IL
```

You can pin a lambda's return type explicitly (C# 10+), which resolves ambiguity and avoids surprising covariance — useful when the inferred type would be too narrow (e.g., you want `Func<object>` not `Func<string>`):

```
var factory = object () => "hello"; // explicit
    return type → Func<object>
```

Default lambda parameters (C# 12) and parameter modifiers (C# 14)

C# 12 allowed lambda parameters to carry default values and `params`, matching what methods could already do:

```
var f = (int x, int step = 1) => x + step;

Console.WriteLine(f(10));    // 11 - uses default
    step
Console.WriteLine(f(10, 5)); // 15
```

The key subtlety: a lambda with a default value (or a `params` parameter) does **not** have `Func<int,int,int>` as its natural type. `Func<>` / `Action<>` cannot express an optional parameter, so the compiler instead synthesizes a brand-new anonymous delegate type (named like `<>f__AnonymousDelegate0`) whose `Invoke` carries the default in its parameter metadata. That synthesized delegate is what `var f` binds to, and that is exactly why `f(10)` can omit `step` — the default lives on the delegate's `Invoke` signature. Assign the lambda to a `Func<int,int,int>` instead (`Func<int,int,int> g = (x, step) => x + step;` — you must drop the default, since `Func` has no slot for it) and both parameters become mandatory: `g(10)` won't compile. The

default is a property of the synthesized delegate type, not a portable feature you can carry onto an arbitrary `Func`.

C# 14 removed the last asymmetry: lambda parameters may now carry `ref`, `out`, `in`, `ref readonly`, and `scoped` modifiers *without an explicit type*, inferring the type from the target delegate. Before C# 14, any modifier forced you to spell the type. This matters for delegates like `TryParse`-shaped signatures and span transforms.

```
// A delegate whose second parameter is 'out'.
delegate bool TryConvert<T>(string text, out T value);

// C# 14: 'out result' carries the modifier but omits
// the type – inferred as 'out int'.
TryConvert<int> tryInt = (text, out result) =>
    int.TryParse(text, out result);

if (tryInt("42", out int n))
    Console.WriteLine(n); // 42
```

A `ref` example, mutating the caller's variable through an implicitly typed lambda parameter:

```
delegate void Accumulate(ref int total, int amount);

// 'ref running' – modifier without a type; compiler
// infers 'ref int' from Accumulate.
Accumulate add = (ref running, amount) => running +=
    amount;

int sum = 0;
add(ref sum, 10);
add(ref sum, 32);
Console.WriteLine(sum); // 42
```

The rules: you may omit the type *only* when a modifier is present and the target type is known, you cannot mix implicitly and explicitly typed parameters in one list, and `params` still requires the explicit type. These lambdas have **no natural type** (modifiers preclude one), so they always require an explicit delegate target — `var` won't work here.

Closure mechanics: display classes and capture cost

The single most performance-relevant fact about lambdas is *what they capture*. A lambda that touches no enclosing state is cheap; one that captures a local turns into a heap allocation. The compiler implements capture by hoisting captured variables into a compiler-generated **display class** (named `<c__DisplayClassN_M` in IL). The lambda becomes an instance method on that class, and the delegate's `Target` points at the display-class instance.

Consider:

```
public Func<int> MakeCounter(int start)
{
    int count = start;           // captured
    return () => ++count;         // mutates
                                // captured state across calls
}
```

The compiler rewrites this roughly as:

```
// Conceptual lowering – not source you write (names
// match the <c__DisplayClassN_M convention):
private sealed class <c__DisplayClass0_0
{
    public int count;
    public int <MakeCounter>b__0() => ++count;
}

public Func<int> MakeCounter(int start)
{
    var locals = new <c__DisplayClass0_0(); // heap
    // allocation #1
    locals.count = start;
    return locals.<MakeCounter>b__0;        //
    // delegate allocation #2 (target = locals)
}
```

Every call to `MakeCounter` allocates a display-class object **and** a delegate. Two `MakeCounter` calls share nothing — each gets its own display instance, which is exactly the closure semantics you want. But when an equivalent capture sits on a hot path (per-ele-

ment in a tight loop, per-request in a server), those two allocations recur every time the enclosing scope is entered — GC pressure you may not have budgeted for.

A subtle and costly variant: the compiler hoists the *entire* set of variables captured by *any* lambda in a scope into a single display class. If a method has two lambdas and one captures `a` while the other captures `b`, both `a` and `b` may land in one shared display object, extending the lifetime of `b` to match `a`. Worse, if a lambda in a loop captures an outer-scope variable, the display class can be allocated *once per method call* and shared, or *once per iteration* — depending on which scope the captured variable lives in. Read the IL when it matters.

Static lambdas: proving the absence of capture

C# 9 introduced the `static` lambda modifier, which is a *compile-time assertion* that the lambda captures nothing from the enclosing scope. If you accidentally reference a local or instance field, you get a compile error instead of a silent allocation. It is the cheapest insurance against accidental closures.

```
int threshold = 10;

// Capturing: 'threshold' is hoisted into a display
// class → allocation.
Func<int, bool> capturing = n => n > threshold;

// Static: cannot see 'threshold'. Pass state
// explicitly instead.
Func<int, int, bool> noCapture = static (n, t) => n >
    t;
```

A `static` lambda that references no state has no `Target`, so the compiler can **cache a single delegate instance** in a static field and reuse it for every evaluation — zero steady-state allocation. The capturing version allocates a fresh display class plus delegate on each call to the enclosing method. Inspecting the IL makes the difference unmistakable:

```

// Capturing version (decompiled essence):
// newobj <>c__DisplayClass.ctor      // heap alloc
//   per call
// stfld  threshold
// ldflld <>9__0 / newobj Func`2      // delegate
//   alloc bound to display class

// Static version:
// ldsfld <>c.<>9__0_0                // load
//   cached delegate from static field
// dup / brtrue (skip init)          // create
//   exactly once, reuse forever
// newobj Func`2 (target = null)      // first
//   call only

```

The pattern to internalize: when a higher-order API offers a state parameter, prefer `static` lambda + passed-in state over a capturing lambda. The BCL added these state-threading overloads precisely to kill closures on hot paths:

```

var cd = new
    System.Collections.Concurrent.ConcurrentDictionary<string, int>();
int factor = 3;

// Capturing closure: the (key, valueFactory)
//   overload's lambda closes over 'factor',
//   so each call site that reaches the factory path
//   allocates a display class + delegate.
int v1 = cd.GetOrAdd("a", key => key.Length * factor);

// State-threaded static lambda: the (key,
//   valueFactory, factoryArgument) overload
//   passes 'factor' through TArg, so the lambda
//   captures nothing and its delegate is cached.
int v2 = cd.GetOrAdd("b", static (key, f) =>
    key.Length * f, factor);

//
//   ^^^^^ state passed by value, not captured

```

`ConcurrentDictionary.GetOrAdd<TArg>(key, valueFactory, factoryArgument)` has the signature `TValue GetOrAdd<TArg>(TKey, Func<TKey,TArg,TValue>, TArg)`. The same state-threading shape appears in `string.Create<TState>(int length, TState state, SpanAction<char,TState>)` and simi-

lar high-performance BCL entry points, all so you can express logic without capturing. Use them.

Method-group conversions and delegate caching

A *method group* — a bare method name used where a delegate is expected — converts to a delegate. Since C# 11, the compiler **caches** the delegate produced from a method-group conversion of a *static* method in a hidden static field, so repeated conversions yield the *same* delegate instance. (The language spec merely *permits* this reuse; Roslyn applies it to static method groups, whose target is `null` and therefore safe to cache globally.

Instance method groups are not cached this way, because the cached delegate would have to bind a specific receiver.)

```
static int Square(int x) => x * x;

Func<int, int> f1 = Square;
Func<int, int> f2 = Square;

Console.WriteLine(ReferenceEquals(f1, f2)); // True -
      cached since C# 11
```

Before C# 11 each conversion allocated a fresh delegate; the cache was a measurable win for LINQ-heavy code that passed the same method group repeatedly (e.g., `.Select(Square)` in a loop). The cache applies to method-group conversions, **not** to lambdas: two textually identical lambdas are two distinct delegates unless they are non-capturing, in which case the compiler may cache each lambda's delegate separately (but the two source expressions still produce two cache slots).

Caveat — instance method groups that capture a receiver are *not* reference-equal across conversions of *different* receivers, and equality compares `(target, method)` :

```

var list = new List<int>();
Action<int> g1 = list.Add;    // List<T>.Add returns
                             void → binds to Action<int> cleanly
Action<int> g2 = list.Add;
Console.WriteLine(ReferenceEquals(g1, g2)); // False
                                           - instance method-group conversions are not
                                           cached
Console.WriteLine(g1 == g2);              // True -
                                           delegate equality compares target + method, so
                                           these are equal

```

(A method group only converts to a delegate whose return type is compatible: `List<int>.Add` returns `void`, so it binds to `Action<int>`. A method that returns a value — e.g. `StringBuilder.Append`, which returns `StringBuilder` — does **not** convert to `Action`; that is a [CS0407](#) compile error, because delegate conversion does not silently discard return values.)

The takeaway: rely on `==` (value equality on target+method) for *deduplication* (e.g., preventing double event subscription), and rely on the method-group *cache* (reference equality, static methods) for *allocation avoidance*. They are different guarantees.

Expression<TDelegate>: code as data

A lambda assigned to `Expression<TDelegate>` is **not compiled to IL**. Instead the compiler emits code that builds an *expression tree* — an object graph of `System.Linq.Expressions` nodes describing the lambda’s structure. This is the mechanism that lets LINQ providers (EF Core, `IQueryable`) translate `o => o.Total > 100` into SQL: they never run your lambda, they *inspect* it.

You can build the same tree by hand. Here is `Expression<Func<Order, bool>>` for “total over 100 and not cancelled”, constructed node by node and then compiled to an executable delegate:

```

using System.Linq.Expressions;

public record Order(decimal Total, bool IsCancelled);

// Build: order => order.Total > 100m &&
//        !order.IsCancelled
ParameterExpression order =
    Expression.Parameter(typeof(Order), "order");

Expression total      = Expression.Property(order,
    nameof(Order.Total));
Expression overHundred = Expression.GreaterThan(total,
    Expression.Constant(100m));

Expression cancelled  = Expression.Property(order,
    nameof(Order.IsCancelled));
Expression notCancelled = Expression.Not(cancelled);

Expression body = Expression.AndAlso(overHundred,
    notCancelled);

Expression<Func<Order, bool>> predicate =
    Expression.Lambda<Func<Order, bool>>(body, order);

Console.WriteLine(predicate);
// order => ((order.Total > 100) AndAlso
//           Not(order.IsCancelled))

Func<Order, bool> compiled = predicate.Compile(); //
// emits IL via the expression compiler
Console.WriteLine(compiled(new Order(150m, false)));
// True
Console.WriteLine(compiled(new Order(150m, true)));
// False

```

`Compile()` is expensive: it walks the tree and emits a `DynamicMethod` with real IL, JIT-compiled on first invocation. The resulting delegate runs at near-hand-written speed, but the *compilation* can cost microseconds to milliseconds. **Cache the compiled delegate; never call `Compile()` per request.** (See the pitfall below.) For startup-sensitive or AOT scenarios, `Compile(preferInterpretation: true)` produces an interpreted delegate that skips JIT — slower per call, far cheaper to create, and AOT-friendly since it emits no runtime IL.

Rewriting trees with ExpressionVisitor

Because expression trees are immutable object graphs, you transform them by *rebuilding*. `ExpressionVisitor` (the BCL base class) implements the visitor pattern: override the `Visit*` methods for node kinds you care about, return new nodes to replace them, and let the base class recurse and reassemble everything else. This is exactly how EF Core and AutoMapper rewrite your queries.

A canonical, genuinely useful task: combine two `Expression<Func<T,bool>>` predicates with `AndAlso`. You cannot just `Expression.AndAlso(a.Body, b.Body)` because each lambda has its *own* `ParameterExpression` instance — the combined tree would reference two different parameters. The fix is a visitor that rewrites every occurrence of `b`'s parameter to `a`'s parameter:

```

using System.Linq.Expressions;

sealed class ParameterReplacer(ParameterExpression
    from, ParameterExpression to)
    : ExpressionVisitor    // C# 12 primary
    constructor
{
    protected override Expression
        VisitParameter(ParameterExpression node)
        => node == from ? to :
        base.VisitParameter(node);
}

static class PredicateExtensions
{
    public static Expression<Func<T, bool>> AndAlso<T>
        (
            this Expression<Func<T, bool>> left,
            Expression<Func<T, bool>> right)
        {
            // Reuse left's parameter; rewrite right's
            // body to reference it.
            ParameterExpression p = left.Parameters[0];
            Expression rewrittenRight =
                new ParameterReplacer(right.Parameters[0],
                    p).Visit(right.Body)!;

            Expression body =
                Expression.AndAlso(left.Body, rewrittenRight);
            return Expression.Lambda<Func<T, bool>>(body,
                p);
        }
}

```

Usage — and proof the combined tree is itself inspectable (so EF Core can still translate it to SQL):

```

Expression<Func<Order, bool>> big      = o => o.Total
    > 100m;
Expression<Func<Order, bool>> notDead  = o =>
    !o.IsCancelled;

Expression<Func<Order, bool>> combined =
    big.AndAlso(notDead);

Console.WriteLine(combined);
// o => ((o.Total > 100) AndAlso Not(o.IsCancelled))

var f = combined.Compile();
Console.WriteLine(f(new Order(150m, false))); // True

```

The single shared parameter `o` is the whole point: an `IQueryable` provider walking this tree sees one parameter and produces valid SQL (`WHERE Total > 100 AND NOT IsCancelled`). Skip the visitor — splice two bodies that reference two different `ParameterExpression` instances — and `Compile()` throws an `InvalidOperationException` (“variable ‘o’ of type ... referenced from scope ”, but it is not defined”), while a real provider fails translation for the same reason.

Compiling for dynamic dispatch — and when to reach for a source generator instead

Compiled expression trees are the classic answer to “I need to call a member I only know by name at runtime, but reflection is too slow in a loop.” A reflective `PropertyInfo.GetValue` boxes and does a virtual dispatch on every call; a compiled accessor is a direct field/property read. Build the accessor *once*, cache it, and you pay reflection cost only at setup:


```

using System.Linq.Expressions;
using System.Collections.Concurrent;

static class FastAccessor
{
    // Cache compiled getters by (type, property) so
    // we Compile() at most once each.
    static readonly ConcurrentDictionary<Type,
        string>, Func<object, object?>> _cache =
        new();

    public static Func<object, object?> GetGetter(Type
        type, string property) =>
        _cache.GetOrAdd((type, property), static key
            =>
            {
                var (t, name) = key;
                ParameterExpression instance =
                    Expression.Parameter(typeof(object), "i");
                Expression typed =
                    Expression.Convert(instance, t);
                Expression access =
                    Expression.Property(typed, name);
                Expression boxed =
                    Expression.Convert(access, typeof(object));
                // box value types
                return Expression.Lambda<Func<object,
                    object?>>(boxed, instance).Compile();
            });
}

// Setup cost paid once; steady-state is a direct
// property read + cache lookup.
var getTotal = FastAccessor.GetGetter(typeof(Order),
    nameof(Order.Total));
Console.WriteLine(getTotal(new Order(99m, false)));
// 99

```

Note the `static` lambda passed to `GetOrAdd` — it captures nothing, so the factory itself allocates no closure; `key` carries all the state.

When to prefer a source generator (Roslyn) instead.

Expression-tree compilation is a *runtime* technique with three liabilities: (1) the first call pays JIT for the emitted

`DynamicMethod`; (2) it is incompatible with Native AOT unless you force `preferInterpretation: true`, and even then it carries a trimming/interpreter cost; (3) errors surface at runtime,

not build time. A **source generator** computes the same accessor *at compile time*, emits ordinary C# that the main compiler and AOT toolchain see, costs zero startup JIT, trims cleanly, and gives you compile-time diagnostics. The modern BCL has shifted decisively this way — `System.Text.Json` source generation, `LoggerMessage` generators, and `[GeneratedRegex]` all replace what used to be reflection-or-expression-tree machinery. Rule of thumb: if the set of types is known at build time, use a source generator; if it is genuinely dynamic (plugin assemblies, user-supplied schemas, `IQueryable` translation), use expression trees and cache aggressively.

Worked example: a tiny rules engine with cached, composable predicates

This ties the chapter together: build predicates as expression trees so they remain *inspectable* (could be pushed to a database via `IQueryable`), compose them with the `AndAlso` visitor, compile **once**, cache the result, and execute with zero per-evaluation allocation thanks to a static lambda in the cache factory.

```

using System.Linq.Expressions;
using System.Collections.Concurrent;

public record Customer(string Country, int
    OrdersLastYear, decimal LifetimeValue);

public sealed class RuleEngine
{
    // Each compiled rule is cached by name; Compile()
    runs at most once per rule.
    readonly ConcurrentDictionary<string,
        Func<Customer, bool>> _compiled = new();
    readonly Dictionary<string,
        Expression<Func<Customer, bool>>> _rules = [];

    public void Define(string name,
        Expression<Func<Customer, bool>> rule) =>
        _rules[name] = rule;

    public bool Evaluate(string name, Customer c) =>
        _compiled.GetOrAdd(name, static (key, rules)
            => rules[key].Compile(), _rules)(c);
    // ^ static lambda, state
    threaded via _rules - no closure allocation
}

// --- composition helper from earlier, reused
verbatim ---
static class Predicates
{
    sealed class Replacer(ParameterExpression from,
        ParameterExpression to) : ExpressionVisitor
    {
        protected override Expression
        VisitParameter(ParameterExpression n) => n ==
            from ? to : n;
    }

    public static Expression<Func<T, bool>> And<T>(
        this Expression<Func<T, bool>> a,
        Expression<Func<T, bool>> b)
    {
        var p = a.Parameters[0];
        var rb = new Replacer(b.Parameters[0],
            p).Visit(b.Body)!;
        return Expression.Lambda<Func<T, bool>>
            (Expression.AndAlso(a.Body, rb), p);
    }
}

class Demo
{

```

```

static void Main()
{
    Expression<Func<Customer, bool>> isEu      = c
    => c.Country == "CZ" || c.Country == "DE";
    Expression<Func<Customer, bool>> isLoyal   = c
    => c.OrdersLastYear >= 12;
    Expression<Func<Customer, bool>> isHighLtv = c
    => c.LifetimeValue > 10_000m;

    var engine = new RuleEngine();
    engine.Define("vip",
        isEu.And(isLoyal).And(isHighLtv));    //
        composed expression tree

    var alice = new Customer("CZ", 15, 25_000m);
    var bob   = new Customer("US", 3, 500m);

    Console.WriteLine(engine.Evaluate("vip",
        alice)); // True - compiles once, caches
    Console.WriteLine(engine.Evaluate("vip",
        bob));   // False - cache hit, no Compile()
    Console.WriteLine(engine.Evaluate("vip",
        alice)); // True - cache hit
}
}

```

Three things make this production-grade: the rules stay as `Expression<>` so they could be handed to EF Core for server-side filtering instead of in-memory evaluation; `Compile()` is guarded by `GetOrAdd` so it runs once per named rule regardless of evaluation frequency; and the cache factory is a `static` lambda with state threaded through `GetOrAdd`'s `TArg` overload, so the hot path allocates nothing beyond the one-time delegate.

Pitfalls and performance notes

Capturing a loop variable. The classic C# bug — closing over a `for`-loop variable and observing every closure see the *final* value — exists because the variable is declared once and shared across iterations. `foreach` was fixed in **C# 5**: the iteration variable is now a fresh binding per iteration, so closures capture distinct values. A `for` loop's variable is still shared:

```

var foreachFns = new List<Func<int>>();
foreach (int i in Enumerable.Range(0, 3))
    foreachFns.Add(() => i);
Console.WriteLine(string.Join(", ", foreachFns.Select(f
    => f()))); // 0,1,2 - fixed since C# 5

var forFns = new List<Func<int>>();
for (int j = 0; j < 3; j++)
    forFns.Add(() => j);
Console.WriteLine(string.Join(", ", forFns.Select(f =>
    f()))); // 3,3,3 - shared variable, all
            see final j

// The fix for 'for': introduce a per-iteration copy.
var fixedFns = new List<Func<int>>();
for (int j = 0; j < 3; j++)
{
    int copy = j; // fresh binding
                  each iteration → captured separately
    fixedFns.Add(() => copy);
}
Console.WriteLine(string.Join(", ", fixedFns.Select(f
    => f()))); // 0,1,2

```

Allocating a closure on a hot path. A capturing lambda allocates a display class and a delegate each time the *enclosing method* runs. Note the precise lifetime: a single `items.Select(p => p * (1 + tax))` builds the closure **once** (the lambda is materialized when `Select` is called, not per element — LINQ enumerates lazily). The pressure shows up when that whole expression sits inside a per-request or per-iteration method, so the one allocation recurs every call. `Enumerable.Select` has **no** (source, selector, state) state-threading overload — that remains an open BCL proposal, not shipped API — so the realistic fixes are (a) a `static` lambda that closes over nothing because its only free name is a `const`, or (b) hoisting the computed delegate into a static field so it is allocated once:

```

const decimal Tax = 0.21m;           // const → a
                                     static lambda can read it without capturing

decimal[] PriceItems(decimal[] items)
{
    // Bad on a hot path: a *non-const* local would
    // force a capturing closure per call.
    // decimal tax = 0.21m;
    // return items.Select(p => p * (1 +
    // tax)).ToArray(); // display class allocated
    // each call

    // Good: static lambda captures nothing (Tax is
    // const); its delegate is cached once.
    return items.Select(static p => p * (1 +
    Tax)).ToArray();
}

// Or, if the multiplier is genuinely a runtime value,
// thread it through a state-carrying
// API that *does* exist, e.g.
// ConcurrentDictionary.GetOrAdd<TArg> or
// string.Create<TState>,
// or precompute and hoist the delegate:
static readonly Func<decimal, decimal> s_addTax =
    static p => p * (1 + Tax); // allocated once
    at type init

```

Compiling expression trees per call. This is the most expensive mistake in the chapter. `expr.Compile()` emits and JITs a `DynamicMethod`; doing it inside the method that uses the delegate throws away that work every call:

```

// CATASTROPHIC on a hot path: Compile() (µs-ms) runs
// on every invocation.
bool IsBig(Order o)
{
    Expression<Func<Order, bool>> e = x => x.Total >
        100m;
    return e.Compile()(o);    // recompiles the same
                             // tree forever
}

// Correct: compile once, cache the delegate in a
// static field.
static readonly Func<Order, bool> s_isBig =
    ((Expression<Func<Order, bool>>)(x => x.Total >
        100m)).Compile();
bool IsBigFast(Order o) => s_isBig(o);    // direct
                                         // call, JIT paid once at type init

```

A final note on the newest surface area. C# 14's user-defined compound assignment operators (`operator +=` , `operator ++` as void-returning *instance* methods) let mutable types mutate in place instead of allocating a new instance per `+=` , which is a different lever on the same problem this chapter keeps circling: allocation on hot paths. They are unrelated to delegates mechanically, but they share the philosophy — give the compiler enough information to avoid an allocation it would otherwise be forced to make. Whether you reach for a `static` lambda, a method-group cache, a cached compiled expression, or an in-place compound operator, the discipline is identical: know exactly where the heap allocation is, and prove it is gone.

Sources: [What's new in C# 14 — Microsoft Learn](#), [Simple lambda parameters with modifiers — Microsoft Learn](#), [User-defined compound assignment operators — Microsoft Learn](#).

12 · Advanced LINQ and Query Composition

LINQ is two things wearing one syntax: an in-memory iterator pipeline over `IEnumerable<T>`, and an expression-tree compiler target over `IQueryable<T>`. The gap between them — where deferred execution defers, where the work *actually* runs, and what each abstraction costs in allocations and JIT-able IL — is where intermediate developers ship subtle bugs and experts win measurable performance. This chapter is about controlling that gap deliberately.

Deferred execution: the model you must hold in your head

A LINQ query expression is not a result; it is a *recipe*. `Where`, `Select`, `Take`, `OrderBy`, `SelectMany`, and friends return a new `IEnumerable<T>` that captures the source and the delegate, and does nothing until something calls `GetEnumerator()` and pulls. The pull is triggered by a `foreach`, or by a *terminal* operator: `ToList`, `ToArray`, `Count`, `First`, `Single`, `Aggregate`, `Sum`, `Any`, `ToDictionary`, and so on.

```
var source = new[] { 1, 2, 3, 4, 5 };
int sideEffects = 0;

IEnumerable<int> query = source.Where(n =>
{
    sideEffects++;           // proves nothing has run
    yet
    return n % 2 == 0;
});

Console.WriteLine(sideEffects); // 0 – Where has not
    pulled anything
var list = query.ToList();      // NOW the predicate
    runs, five times
Console.WriteLine(sideEffects); // 5
```


This laziness is the whole point of LINQ, but it has two faces. A **streaming** operator yields each element as soon as it can:

`Where`, `Select`, `Take`, `Skip`, `SelectMany`, `Cast`, `Concat`, `Zip`. It holds essentially no state and produces output incrementally — you can run an infinite source through `Select(...).Take(10)` and it terminates. A **fully buffering** operator must consume the *entire* source before it can yield its first element, because its output depends on all the input: `OrderBy` / `OrderByDescending` (it sorts), `Reverse`, `GroupBy`, and `ToLookup`. These allocate a backing collection sized to the input and are where a “lazy” query quietly becomes O(n) memory.

A third group is easy to misfile: `Distinct` and `Union stream` their output — the first distinct element is yielded immediately — but they carry a growing `HashSet` of already-seen elements, so they are streaming in latency yet O(distinct) in memory.

`Except` and `Intersect` are *partially* buffering: they eagerly drain the **second** sequence into a `HashSet` up front, then stream the first sequence against it — so they buffer one input, not both, and start yielding from the first source as soon as the set is built. Knowing which of the three categories an operator lands in is what lets you predict both its first-element latency and its peak memory.

```
// Streaming: pulls at most 3 from a 10-million-
//           element source, 0(1) extra memory
var firstThreeEvens = Enumerable.Range(0, 10_000_000)
    .Where(n => n % 2 == 0)
    .Take(3)
    .ToList(); // [0, 2, 4] — only ~5 elements ever
                enumerated

// Buffering: OrderBy must materialize and sort all
//           10M before yielding one
var sorted = Enumerable.Range(0, 10_000_000)
    .OrderBy(n => n) // allocates a buffer of
                    10M ints + sort
    .Take(3)        // Take cannot help OrderBy
                    avoid the buffer
    .ToList();
```

The `OrderBy().Take(n)` and `OrderByDescending().First()` cases are so common that `System.Linq` special-cases them: the ordered-enumerable implementation runs a *partial* quicksort (and a quickselect for the single-element `First / ElementAt` case) that only fully orders the prefix you actually pull, instead of sorting the whole sequence. But the *buffer* still exists — the source is materialized into an array before the partial sort runs. The lesson: know which operators are buffering, because that is where your memory profile lives.

The double-enumeration bug

Because a query is a recipe, re-enumerating it re-runs it. With pure CPU work that just wastes cycles. With I/O it duplicates network calls, database round-trips, or file reads — and if the source is single-pass (a network stream, a `yield`-based reader), the second enumeration throws or returns empty.

```
IEnumerable<Customer> ExpensiveQuery(DbContext db) =>
    db.Customers.Where(c => c.IsActive).AsEnumerable()
    // pretend: hits the DB
    .Select(c => { Console.WriteLine($"materialized
    {c.Id}"); return c; });

void Report(DbContext db)
{
    var actives = ExpensiveQuery(db);

    int count = actives.Count();           //
    // enumeration #1 - full query runs
    decimal total = actives.Sum(c => c.Ltv); //
    // enumeration #2 - runs AGAIN
    var top = actives.OrderByDescending(c =>
        c.Ltv).First(); // #3

    Console.WriteLine($"{count} actives, {total:C},
        top {top.Id}");
}
```

That method runs the query three times. Each `Count`, `Sum`, and `OrderByDescending().First()` is an independent pull. If `ExpensiveQuery` issues a SQL query or an HTTP request, you just tripled the load — and the bug is invisible in code review be-

cause every line looks idiomatic. The compiler will not warn you; the built-in .NET analyzer rule [CA1851](#) (“Possible multiple enumerations of an `IEnumerable` collection,” shipped with the .NET 7+ SDK and enabled by ruleset packs like Meziantou’s and Roslynator) exists precisely because the language cannot. It uses dataflow analysis and misses some cases, so it is a backstop, not a guarantee.

The fix is to materialize once at the boundary:

```
void Report(DbContext db)
{
    List<Customer> actives =
        ExpensiveQuery(db).ToList();    // one
        enumeration

    int count = actives.Count;
    // List.Count – O(1) property
    decimal total = actives.Sum(c => c.Ltv);
    // over the in-memory list
    var top = actives.MaxBy(c => c.Ltv);
    // MaxBy (.NET 6) avoids a sort
    Console.WriteLine($"{count} actives, {total:C},
        top {top.Id}");
}
```

Note the secondary win: once you hold a `List<T>`, `Count` is the `O(1) List<T>.Count` property, not the `Enumerable.Count()` extension method that walks the sequence. And `MaxBy` (introduced in .NET 6) replaces `OrderByDescending().First()` with a single `O(n)` scan and no sort buffer. Materialization is not free — you pay for the list — but you pay *once*, deterministically, instead of an unbounded number of times depending on how many terminal operators a caller happens to chain.

When measured: re-running the three-enumeration version against an in-memory list of 1,000,000 records under `BenchmarkDotNet` shows $\sim 3\times$ the element-visit count of the materialized version and proportional time; against a real database the difference is three round-trips versus one, which dwarfs any CPU consideration. The rule of thumb: **an `IEnumerable<T>` returned across an API boundary that will be consumed more than once should be materialized, or the bound-**

ary should return `ReadOnlyList<T>` to make single-pass-vs-reusable explicit in the type.

`IEnumerable<T>` VS `IQueryable<T>` : where the work runs

This is the most consequential distinction in LINQ.

`IEnumerable<T>` carries *delegates* — compiled `Func<T, bool>` — and executes them in your process, in IL, pulling element by element. `IQueryable<T>` carries *expression trees* —

`Expression<Func<T, bool>>`, a data structure describing the lambda — which a provider (EF Core, a remote OData client) translates to another language (SQL) and executes elsewhere.

```
// IQueryable: the predicate is an Expression tree,
// translated to SQL.
// WHERE [c].[IsActive] = 1 AND [c].[Region] = N'EU'
// runs in the database.
IQueryable<Customer> q1 = db.Customers
    .Where(c => c.IsActive && c.Region == "EU");

// IEnumerable: AsEnumerable() switches to in-memory.
// Everything after it
// is a compiled delegate running in THIS process —
// the DB returns all rows.
IEnumerable<Customer> q2 = db.Customers
    .AsEnumerable() //
    .Where(c => c.IsActive && Score(c)); // Score()
// runs locally, per row
```

The boundary in `q2` is a footgun. The instant you call `AsEnumerable()` (or `ToList()`, or `iterate()`), the provider materializes everything *up to that point* and the rest runs client-side. Put it too early and you drag every row across the wire to filter three of them in memory.

The subtler version is **silent client evaluation**. If you call a method the provider cannot translate inside an `IQueryable` chain, older EF (EF6) would silently evaluate it client-side; EF Core deliberately **throws** for the top-level `Where` / `OrderBy` since EF Core 3.0 rather than hide the perf cliff:

```

static bool IsPremium(Customer c) => c.Ltv > 10_000m;
    // a local method

// THROWS at translation time (EF Core 3.0+):
// IsPremium can't become SQL.
// InvalidOperationException: could not be translated.
var bad = db.Customers.Where(c =>
    IsPremium(c)).ToList();

// Correct: inline the logic so the expression tree is
// fully translatable.
var good = db.Customers.Where(c => c.Ltv >
    10_000m).ToList();

```

EF Core *will* still evaluate constant subexpressions and projections client-side in places it considers safe, and it is easy to write a `Select` that pulls a whole entity graph just to call `.ToString()` locally. The discipline: treat the `IQueryable` portion of a query as “code that must compile to SQL,” and explicitly mark the transition to in-memory with `AsEnumerable()` *after* the database has done all the filtering and projection it can. Project to the narrow shape you need (`Select` into a DTO) *before* the boundary so the wire payload is minimal.

Composing `IQueryable` predicates and projections safely

Building filters at runtime — a search screen with optional criteria — is where expression composition earns its keep. The wrong way is to branch into `IEnumerable` and lose translation:

```

// WRONG when 'predicate' captures runtime conditions
// you build with || in C#:
// if you compose with C# delegate combinators, you're
// back to client eval.

```

The right way keeps everything as `Expression<Func<T, bool>>` and lets EF Core translate the combined tree. The simplest composition is just chaining `Where` — EF Core ANDs them in SQL:

```

IEnumerable<Customer> Build(IEnumerable<Customer> q,
    CustomerFilter f)
{
    if (f.ActiveOnly)           q = q.Where(c =>
        c.IsActive);
    if (f.Region is not null)   q = q.Where(c =>
        c.Region == f.Region);
    if (f.MinLtv is decimal min) q = q.Where(c =>
        c.Ltv >= min);
    return q;                   // all three become
                                one WHERE ... AND ... AND ...
}

```

Chained `Where` only gives you AND. For OR, or to combine two predicates you were *handed* as expressions, you compose the trees by hand. The mechanics: take two `Expression<Func<T, bool>>`, rewrite the second's parameter to reuse the first's, and join the bodies with `Expression.AndAlso` / `OrElse` :

```

using System.Linq.Expressions;

static class PredicateComposer
{
    public static Expression<Func<T, bool>> And<T>(
        this Expression<Func<T, bool>> left,
        Expression<Func<T, bool>> right) =>
        Combine(left, right, Expression.AndAlso);

    public static Expression<Func<T, bool>> Or<T>(
        this Expression<Func<T, bool>> left,
        Expression<Func<T, bool>> right) =>
        Combine(left, right, Expression.OrElse);

    private static Expression<Func<T, bool>>
        Combine<T>(
            Expression<Func<T, bool>> left,
            Expression<Func<T, bool>> right,
            Func<Expression, Expression, BinaryExpression>
            join)
    {
        // Reuse left's parameter; rewrite every
        reference to right's parameter.
        ParameterExpression p = left.Parameters[0];
        Expression rewrittenRight = new
            ReplaceParameter(right.Parameters[0], p)
                .Visit(right.Body);
        return Expression.Lambda<Func<T, bool>>(
            join(left.Body, rewrittenRight), p);
    }

    private sealed class
        ReplaceParameter(ParameterExpression from,
            ParameterExpression to)
        : ExpressionVisitor
    {
        // Primary constructor (C# 12): captures
        from/to directly into the visitor.
        protected override Expression
            VisitParameter(ParameterExpression node)
            => node == from ? to :
            base.VisitParameter(node);
    }
}

```

`ReplaceParameter` uses a **primary constructor** (introduced in C# 12) so the visitor needs no boilerplate field declarations. The critical subtlety is the parameter rewrite: each lambda declares its *own* `ParameterExpression`, and an expression tree

referencing a parameter not in its lambda's parameter list is invalid and EF Core will reject it. The visitor walks `right.Body` and swaps `right`'s parameter for `left`'s, producing a single coherent tree.

```
Expression<Func<Customer, bool>> isEu = c => c.Region
    == "EU";
Expression<Func<Customer, bool>> isVip = c => c.Ltv >
    50_000m;

// Translates to: WHERE [c].[Region] = N'EU' OR [c].
//               [Ltv] > 50000.0
List<Customer> hits =
    db.Customers.Where(isEu.Or(isVip)).ToList();
```

Because the result is still an `Expression<Func<T, bool>>`, EF Core sees one tree and emits one SQL `WHERE`. No client evaluation, no extra round-trips. (The well-known LINQKit library generalizes this with `PredicateBuilder` and an `AsExpandable()` hook that also lets you *invoke* one expression inside another; the hand-rolled version above is enough when you control both predicates.)

Writing custom LINQ operators: iterators and allocation awareness

A LINQ operator is just an extension method on `IEnumerable<T>` that returns `IEnumerable<TResult>`. The iterator (`yield return`) is the idiomatic implementation and gives you deferred, streaming execution for free — the C# compiler generates a state machine class implementing `IEnumerator<T>`.


```

public static IEnumerable<T> WhereNotNull<T>(this
    IEnumerable<T?> source)
    where T : class
{
    ArgumentNullException.ThrowIfNull(source); //
        runs eagerly? No – see below.
    foreach (T? item in source)
        if (item is not null)
            yield return item; //
                streaming, lazy
}

```

There is a real gotcha here: **argument validation in an iterator method is deferred**, because the *entire body* — including the `ThrowIfNull` — does not run until the first `MoveNext()`. If you want eager validation (so a null source throws at the call site, not at the first `foreach`), split the public method from a private iterator:

```

public static IEnumerable<T> WhereNotNull<T>(this
    IEnumerable<T?> source)
    where T : class
{
    ArgumentNullException.ThrowIfNull(source); //
        eager: runs at call time
    return Iterator(source);

    static IEnumerable<T> Iterator(IEnumerable<T?>
        src) // local function iterator
    {
        foreach (T? item in src)
            if (item is not null)
                yield return item;
    }
}

```

The local-function iterator pattern (the `static` local function avoids capturing) is the canonical fix.

Fusing two operators into a single pass

Now the allocation argument.

`source.Where(pred).Select(proj)` allocates *two* iterator state machines and chains two `MoveNext()` calls per element. For a

one-off query that is irrelevant. In a hot path it is two heap objects and an extra layer of virtual dispatch per element. A fused `WhereSelect` operator does the filter and projection in one state machine:

```
public static class FusedOperators
{
    /// <summary>Filter then project in a single
    /// iterator – one state machine,
    /// one MoveNext chain, instead of
    /// Where(...).Select(...)'s two.</summary>
    public static IEnumerable<TResult>
        WhereSelect<TSource, TResult>(
            this IEnumerable<TSource> source,
            Func<TSource, bool> predicate,
            Func<TSource, TResult> selector)
    {
        ArgumentNullException.ThrowIfNull(source);
        ArgumentNullException.ThrowIfNull(predicate);
        ArgumentNullException.ThrowIfNull(selector);
        return Iterator(source, predicate, selector);

        static IEnumerable<TResult> Iterator(
            IEnumerable<TSource> src,
            Func<TSource, bool> pred,
            Func<TSource, TResult> sel)
        {
            foreach (TSource item in src)
                if (pred(item))
                    yield return sel(item);    //
            // filter + map in one pull
        }
    }
}
```

```
// Two iterators, two state-machine allocations:
var a = orders.Where(o => o.Total > 100m).Select(o =>
    o.Id);

// One iterator, one allocation, one MoveNext chain
// per element:
var b = orders.WhereSelect(o => o.Total > 100m, o =>
    o.Id);
```

What does the BCL actually do? `System.Linq` is smarter than this naive picture for the *common* path: when the source is an

array or a `List<T>`, the `Where` → `Select` combination is internally fused by specialized hidden classes (`WhereSelectArrayIterator`, `WhereSelectListIterator`) that combine the predicate and selector into one iterator — so `Where(...).Select(...)` over a `List<T>` is *already* a single fused iterator with no double-machine cost. (`Where` alone over the same sources gets its own `WhereArrayIterator` / `WhereListIterator`; the fusion kicks in when a `Select` is appended.) Your hand-rolled `WhereSelect` only beats the BCL when the source is a *non-specialized* `IEnumerable<T>` (a `yield` method, a `Concat`, a custom collection), where the BCL falls back to generic chaining. **Measure before you fuse:** against a `List<T>` source, custom `WhereSelect` is a wash with the BCL; against an opaque `IEnumerable<T>`, it saves one allocation and one indirection per query (not per element — the state machine is allocated once). The per-element savings is the eliminated second `MoveNext` virtual call, which BenchmarkDotNet will show as single-digit-percent on tight loops and noise on anything that does real work per element.

The newer BCL operators: `Chunk`, `CountBy`, `AggregateBy`, `Index`

Several operators added in recent releases replace whole pipelines.

`Chunk<T>(int size)` (introduced in .NET 6) batches a sequence into `T[]` arrays of at most `size` — the right tool for paging an enumerable into bulk inserts or API calls without a hand-rolled buffer.

```
foreach (int[] batch in ids.Chunk(500))    // .NET 6
    await api.BulkFetchAsync(batch);        // 500
    ids per call
```

`CountBy` and `AggregateBy` (both introduced in .NET 9 / C# 13) collapse the extremely common `GroupBy(...).Select(g => ...)` pattern into a single streaming-friendly pass that never

materializes the per-group `IGrouping<TKey, TElement>` lists.
`CountBy` returns `IEnumerable<KeyValuePair<TKey, int>>` ;
`AggregateBy` returns
`IEnumerable<KeyValuePair<TKey, TAccumulate>>` .

```
record Sale(string Region, string Product, decimal
            Amount);

Sale[] sales =
[
    new("EU", "A", 100m), new("EU", "B", 250m),
    new("US", "A", 400m), new("EU", "A", 50m),
    new("US", "B", 75m),
];

// OLD: GroupBy materializes a list of Sale per
//      region, then counts/sums it.
var oldCounts = sales
    .GroupBy(s => s.Region)
    .Select(g => new { Region = g.Key, Count =
        g.Count() });

var oldTotals = sales
    .GroupBy(s => s.Region)
    .Select(g => new { Region = g.Key, Total = g.Sum(s
        => s.Amount) });

// NEW (.NET 9): CountBy – one pass, no per-group
//              buffers.
// => [ {EU, 3}, {US, 2} ]
IEnumerable<KeyValuePair<string, int>> counts =
    sales.CountBy(s => s.Region);

// NEW (.NET 9): AggregateBy – per-key fold with a
//              seed. seed 0m, add Amount.
// => [ {EU, 400}, {US, 475} ]
IEnumerable<KeyValuePair<string, decimal>> totals =
    sales.AggregateBy(
        keySelector: s => s.Region,
        seed: 0m,
        func: (acc, s) => acc + s.Amount);

foreach (var (region, total) in totals)
    Console.WriteLine($"{region}: {total:C}");
```

`AggregateBy` has two seed overloads: a constant `seed` (`TAccumulate`) as above, and a `seedSelector` (`Func<TKey, TAccumulate>`) when each group needs a fresh mutable accumu-

lator (e.g. a per-key `List<T>` or `StringBuilder`) — use the selector form to avoid sharing one seed instance across keys:

```
// seedSelector form: each region gets its OWN list –  
    never share a mutable seed.  
IEnumerable<KeyValuePair<string, List<string>>>  
    productsByRegion = sales.AggregateBy(  
    keySelector: s => s.Region,  
    seedSelector: _ => new List<string>(),  
    func: (list, s) => { list.Add(s.Product); return  
        list; });
```

The win is not just brevity. `GroupBy` buffers every element into per-key lists before you can reduce them;

`CountBy` / `AggregateBy` keep only the running accumulator per key in a single dictionary, so peak memory is $O(\text{distinct keys})$ instead of $O(\text{elements})$. For a 10-million-row sum over 50 regions, that is 50 decimals of state versus 10 million buffered `Sale` references.

`Index()` (introduced in **.NET 9**) yields `(int Index, TSource Item)` tuples — a clean replacement for `Select((item, i) => ...)` and the C-style `for` loop when you need the position:

```
foreach ((int i, string line) in  
    File.ReadLines(path).Index()) // .NET 9  
    Console.WriteLine($"{i,4}: {line}");
```

SelectMany and flattening

`SelectMany` flattens a sequence-of-sequences into one stream. It is the LINQ equivalent of a nested loop, and reads better than one when the outer/inner relationship is a projection:

```

record Order(int Id, IReadOnlyList<LineItem> Lines);
record LineItem(string Sku, int Qty);

// Nested loops:
var flat1 = new List<(int OrderId, string Sku)>();
foreach (var o in orders)
    foreach (var li in o.Lines)
        flat1.Add((o.Id, li.Sku));

// SelectMany with a result selector – flattens AND
// keeps the outer context:
IEnumerable<(int OrderId, string Sku)> flat2 = orders
    .SelectMany(o => o.Lines, (o, li) => (o.Id,
        li.Sku));

```

The two-delegate overload (collection selector + result selector) is the one to reach for: it gives you both the outer and inner element so you do not lose the parent, and it streams — no intermediate `List`. Where `SelectMany` does *not* win is when the inner work has side effects or early-exit logic that reads more naturally as imperative code; LINQ has no `break`, only `Take` / `TakeWhile`, and forcing complex control flow through query operators hurts readability more than the nested loop ever did.

Span-friendly alternatives on hot paths

LINQ's cost model is: a state-machine allocation per operator (mitigated by BCL fusion for arrays/lists), a delegate invocation per element (not inlined across the iterator boundary), and `IEnumerator<T>` boxing if the source enumerator is a struct exposed through the interface. None of that matters for a query that runs once over a few thousand items. All of it matters in a per-frame, per-request, or per-tick hot loop.

When the source is contiguous — an array, a `List<T>` (via `CollectionsMarshal.AsSpan`), or a stack-allocated buffer — rewriting the LINQ pipeline as a `Span<T>` loop eliminates every allocation and lets the JIT inline the body and autovectorize:

```

using System.Runtime.InteropServices;

// LINQ version – allocates iterator(s), delegates,
// boxes the enumerator:
static decimal SumLargeOrdersLinq(List<Order> orders)
=>
    orders.Where(o => o.Total > 1000m).Sum(o =>
        o.Total);

// Span version – zero allocation, JIT-inlinable,
// cache-friendly:
static decimal SumLargeOrdersSpan(List<Order> orders)
{
    ReadOnlySpan<Order> span =
        CollectionsMarshal.AsSpan(orders); // .NET
        5+, no copy
    decimal total = 0m;
    foreach (Order o in span) // foreach
        over Span is a by-ref index loop
        if (o.Total > 1000m)
            total += o.Total;
    return total;
}

```

`CollectionsMarshal.AsSpan` (introduced in .NET 5) hands you a `Span<T>` directly over the `List<T>`'s internal array — no copy. The `foreach` over a `Span<T>` compiles to an indexed `for` with bounds-check elision, not an `IEnumerator` pull, so there is no per-element virtual call and no enumerator object. Under BenchmarkDotNet over a `List<Order>` of 1,000,000 elements, the span loop typically runs ~2–4× faster than the LINQ pipeline and reports **0 B allocated** versus the LINQ version's iterator allocations — the exact multiple depends on how much real work the body does (a trivial sum shows the largest gap; a body doing string formatting per element narrows it because the per-element work dominates).

C# 14 / .NET 10 sharpens this further with **first-class implicit `Span<T>` / `ReadOnlySpan<T>` conversions**: an array argument now binds to a `ReadOnlySpan<T>` parameter without `.AsSpan()`, so span-first APIs compose with array call sites more cleanly. And **params collections** (introduced in C# 13) let you write `params ReadOnlySpan<T>` parameters that callers

fill with `[a, b, c]` collection-expression syntax with no array allocation at the call site:

```
// C# 13 params ReadOnlySpan<T>: callers pass [a,b,c]
// with no heap array.
static int Max(params ReadOnlySpan<int> values) //
    stack-allocated by the compiler
{
    int max = int.MinValue;
    foreach (int v in values) if (v > max) max = v;
    return max;
}

int m = Max(3, 7, 1, 9, 2); // no int[] allocated
// for the params
```

The caveat: `Span<T>` is a `ref struct`. It cannot be a field of a class, cannot be captured by a lambda, cannot cross an `await` or `yield`, and cannot be a generic type argument. So you cannot “LINQ over a span” in the classic sense — the span loop is the alternative to LINQ, not a backing for it. Reserve it for measured hot paths; in cold code the LINQ pipeline is more readable and the allocations are noise.

LINQ and async: `System.Linq.AsyncEnumerable` in .NET 10

`IAsyncEnumerable<T>` (introduced in C# 8) gives you `await foreach` over a stream that produces elements asynchronously — paged API results, a streaming query, a message queue. Historically, LINQ operators over it required the community `System.Linq.Async` (Ix.NET) NuGet package. **As of .NET 10, the BCL ships `System.Linq.AsyncEnumerable`** with the full standard LINQ operator surface for `IAsyncEnumerable<T>` built in — `Where`, `Select`, `SelectMany`, `Take`, `OrderBy`, `GroupBy`, and async terminal operators `ToListAsync`, `CountAsync`, `FirstAsync`, `AnyAsync`, `SumAsync`, and so on. No package reference needed when targeting `net10.0`.


```

using System.Linq;    // AsyncEnumerable lives here in
                      // .NET 10

async IEnumerable<Page> FetchPagesAsync(
    [System.Runtime.CompilerServices.EnumeratorCan
    cellation] CancellationToken ct = default)
{
    for (int page = 0; ; page++)
    {
        Page p = await http.GetPageAsync(page, ct);
        yield return p;
        if (p.IsLast) yield break;
    }
}

async Task<int>
    CountActiveUsersAsync(CancellationToken ct)
{
    // Streaming async LINQ:
    // Where/SelectMany/CountAsync compose over the
    // pages.
    // Each operator awaits the source as it pulls;
    // nothing buffers the whole stream.
    return await FetchPagesAsync(ct)
        .SelectMany(p => p.Users.ToAsyncEnumerable())
        // flatten pages -> users
        .Where(u => u.IsActive)
        .CountAsync(ct);
    // terminal: awaits to the end
}

```

Two migration-critical details for the .NET 10 BCL implementation, distinct from the old Ix.NET package:

1. Async predicate/selector overloads take a

CancellationToken and use the **base operator name**

— there are no

`WhereAwait` / `SelectAwait` / `AnyAwaitAsync` suffixed variants. Where Ix.NET had `e.SelectAwait(async i => ...)`, the BCL has an overload of `Select` itself accepting `Func<T, CancellationToken, ValueTask<TResult>>`:

```
// .NET 10 BCL: async selector is an overload of
//      Select, with a CancellationToken.
var doubled = source.Select(async (int i,
    CancellationToken ct) =>
{
    await Task.Yield();
    return i * 2;
});
```

2. Terminal operators are suffixed `Async` and accept a `CancellationToken` (`ToListAsync(ct)` , `CountAsync(ct)` , `FirstOrDefaultAsync(ct)`), and a plain `IEnumerable<T>` lifts into the async world via `ToAsyncEnumerable()` .

If you previously referenced the `System.Linq.Async` package and upgrade to `net10.0` , **remove that package reference** — keeping both produces `CS0121` ambiguous-call errors because every operator now exists in two assemblies. (To multi-target older frameworks, reference the `System.Linq.AsyncEnumerable` NuGet package, which back-ports the BCL surface.)

The deferred/streaming/buffering mental model carries over unchanged: `WhereAsync` -style filters stream, `OrderByAsync` and `GroupByAsync` buffer the whole async sequence before yielding, and a double-enumeration of an `IAsyncEnumerable<T>` re-issues every awaited call — so the same `ToListAsync()` -at-the-boundary discipline applies, with the extra teeth that re-awaiting a network stream is far more expensive than re-walking a list.

Worked example: tying it together

A reporting endpoint that (1) builds a translatable `IQueryable` filter at runtime, (2) projects to a narrow DTO *before* the database boundary, (3) materializes once to avoid double-enumeration, and (4) aggregates in-memory with the new `AggregateBy` / `CountBy` operators instead of `GroupBy` :

```

using System.Linq;
using System.Linq.Expressions;
using Microsoft.EntityFrameworkCore; // ToListAsync
    on IQueryable<T> lives here

record SaleRow(string Region, string Product, decimal
    Amount, bool Refunded);
record RegionAmount(string Region, decimal Amount);
    // narrow projection shape
record RegionStat(string Region, int Count, decimal
    Net);

static async Task<IReadOnlyList<RegionStat>>
    RegionReportAsync(
        IQueryable<SaleRow> sales, CustomerFilter f,
        CancellationToken ct)
{
    // 1. Build the predicate as a composed Expression
    // tree – stays translatable.
    Expression<Func<SaleRow, bool>> predicate = s =>
        !s.Refunded; // base condition
    if (f.Region is not null)
        predicate = predicate.And(s => s.Region ==
            f.Region); // composer from earlier
    if (f.MinLtv is decimal min)
        predicate = predicate.And(s => s.Amount >=
            min);

    // 2. Filter and PROJECT in SQL: pull only the two
    // columns the aggregation
    // needs, not whole entities. The Where is one
    // translated WHERE clause.
    List<RegionAmount> rows = await sales
        .Where(predicate)
        .Select(s => new RegionAmount(s.Region,
            s.Amount))
        .ToListAsync(ct); // 3. ONE
    // materialization – no double-enumeration

    // 4. In-memory aggregation with .NET 9 operators
    // – O(distinct regions) state,
    // no per-group buffers (unlike GroupBy),
    // single pass each.
    var counts = rows.CountBy(r => r.Region)
        .ToDictionary(kv => kv.Key, kv =>
            kv.Value);

    return rows
        .AggregateBy(
            keySelector: r => r.Region,
            seed: 0m,
            func: (net, r) => net + r.Amount)

```

```

        .Select(kv => new RegionStat(kv.Key,
            counts[kv.Key], kv.Value))
        .OrderByDescending(stat => stat.Net)    //
            buffering – but over distinct regions only
        .ToList();
    }

```

Every decision in that method is a chapter theme: the `Expression`-tree filter runs in the database and never falls back to client evaluation; the `Select` projection keeps the wire payload to the two columns the report actually consumes; the single `ToListAsync` eliminates the double-enumeration class of bug; `CountBy` and `AggregateBy` replace two `GroupBy` passes with two single-pass folds whose memory is bounded by the region count, not the row count; and the final `OrderByDescending` is the *one* buffering operator in the pipeline, deliberately placed where it sorts a handful of regions rather than millions of rows.

When the abstraction cost is real — and when it isn't

The honest benchmarking conclusion: LINQ's overhead is **per-operator allocation plus per-element delegate dispatch**, and whether that registers depends entirely on the ratio of framework overhead to useful work. A query that hits a database, formats strings, or does any non-trivial per-element computation is dominated by that work — LINQ's overhead is in the noise, and the readability is worth far more than the nanoseconds. A query in a per-frame game loop, a packet decoder, or a numeric kernel over a contiguous buffer, doing a cheap operation millions of times per second, is exactly where the allocation and dispatch dominate — and where the `Span<T>` rewrite, the `params ReadOnlySpan<T>` API, and the fused custom operator pay back.

Three rules survive every benchmark:

- **Materialize at boundaries** to kill double-enumeration — it is correctness, not just speed, when I/O is involved.

- **Keep the `IQueryable` portion translatable**; mark the transition to in-memory explicitly with `AsEnumerable()` after the provider has done all it can, and project narrow before you cross.
- **Reach for `Span<T>` only on measured hot paths over contiguous memory** — and leave LINQ in place everywhere else, because in cold code the abstraction is free and the clarity is not.

The newer operators — `Chunk` (.NET 6), `CountBy` / `AggregateBy` / `Index` (.NET 9), and the built-in `System.Linq.AsyncEnumerable` (.NET 10) — are not syntactic sugar; each one removes a buffering step, an allocation, or a package dependency that the older pipeline forced on you. Use them, attribute them correctly, and benchmark the rest.

**Part IV — High-
Performance C**

13 · Span, Memory, ref structs, and ref Safety

`Span<T>` is the type that finally made it possible to write allocation-free, bounds-checked code that flows uniformly over managed arrays, the stack, and unmanaged memory. But it pays for that power with the most restrictive lifetime model in the language — the `ref struct` — and a ref-safety analysis the compiler runs on every method that touches one. This chapter is about working *with* that model rather than fighting it: when a `Span<T>` is correct, when `Memory<T>` is the only legal choice, and how C# 13’s `allows ref struct` and C# 14’s implicit span conversions remove most of the friction that used to surround them.

The shape of the problem

Before spans, “give me a window into the middle of this buffer” had exactly two answers, both bad. You either allocated — `array.Skip(4).Take(8).ToArray()`, `substring`, a fresh `byte[]` — or you passed `(buffer, offset, count)` triples by hand and prayed every call site got the arithmetic right. The first answer churns the GC; the second is a correctness minefield and composes terribly.

`Span<T>` is a single value type that encodes exactly that triple — a reference to the first element and a length — with the bounds check baked into every indexer access. Crucially it is a *reference into existing memory*, not a copy:

```
int[] data = [10, 20, 30, 40, 50];
Span<int> middle = data.AsSpan(1, 3); // {20, 30, 40}
    - no allocation
middle[0] = 99;                       // writes
    THROUGH to data[1]
Console.WriteLine(data[1]);           // 99
```

The same `Span<int>` type can point at a stack buffer or native memory; the consumer never knows or cares:

```
Span<int> onStack = stackalloc int[5];           // lives
                                                    in the current frame
Span<byte> native = MemoryMarshal.CreateSpan( // wraps
    unmanaged memory
    ref Unsafe.AsRef<byte>((void*)ptr), length);
```

One abstraction, three memory sources, zero allocations, full bounds safety. That is the whole pitch. The cost is that to make a span pointing into stack memory *safe*, the type has to be forbidden from outliving the stack frame it was created in — and enforcing that is what drives everything else in this chapter.

`Span<T>` and `ReadOnlySpan<T>` : the slice abstraction

`Span<T>` is mutable; `ReadOnlySpan<T>` is the read-only projection. Default to `ReadOnlySpan<T>` in API signatures — it accepts everything a `Span<T>` does *plus* `string` and immutable data, and it documents that you won't mutate the caller's buffer. There is an implicit conversion from `Span<T>` to `ReadOnlySpan<T>`, never the reverse.

Both are defined (roughly) as:

```
public readonly ref struct Span<T>
{
    internal readonly ref T _reference; // a ref field
    - the interior pointer
    private readonly int _length;
    public ref T this[int index] => ref ...; //
    bounds-checked, returns a ref
    public int Length { get; }
    public Span<T> Slice(int start, int length);
}
```

That `ref T` field is the heart of it and the reason `Span<T>` must be a `ref struct` (more on that shortly). The indexer returns `ref T`, not `T`, which is why writes go through to the underlying storage and why you can take the address of an element.

Slicing is free. `Slice` (and the range indexer, which lowers to it) produces a new span over the same memory — it copies two fields, the reference and a length, and allocates nothing:

```
ReadOnlySpan<char> s = "key=value";
int eq = s.IndexOf('=');
ReadOnlySpan<char> key = s[..eq];           // "key"    -
                                     range indexer → Slice
ReadOnlySpan<char> val = s[(eq + 1)..]; // "value"
```

`string.IndexOf` returns the position; `s[..eq]` and `s[(eq+1)..]` are zero-allocation windows. Compare this to `string.Split('=')`, which allocates an array *plus* two sub-string objects. For hot parsing paths this difference dominates.

Bounds and ranges — the gotchas. Slicing validates against length, and an off-by-one is an exception, not silent corruption:

```
ReadOnlySpan<int> sp = stackalloc int[] { 1, 2, 3 };
_ = sp.Slice(2, 2); // ArgumentOutOfRangeException:
                  start(2)+length(2) > 3
_ = sp[1..4];      // same - the range's end (4)
                  exceeds Length (3)
```

The from-end index `^` is a frequent source of off-by-one bugs. `^0` is *past the last element* — it equals `Length`, so `sp[^0]` throws, while `sp[^1]` is the last element. `sp[..^1]` drops the last element; `sp[1..^1]` trims one from each end. Read `^n` as “Length – n” and these stop surprising you.

`params ReadOnlySpan<T>` (C# 13)

C# 13 extended `params` to spans (and to any collection type). A `params ReadOnlySpan<T>` parameter lets callers pass a comma-separated list with no array allocation — the compiler stack-allocates the backing storage when it can prove the span doesn’t escape:

```
static int Sum(params ReadOnlySpan<int> values)
{
    int total = 0;
    foreach (int v in values) total += v;
    return total;
}

int t = Sum(1, 2, 3, 4); // no int[] allocated;
                        // backing buffer is on the stack
```

Pre-C# 13 `params int[]` would heap-allocate an array on every call. Most BCL `params` overloads (e.g. `string.Format` -style hot paths) gained `params ReadOnlySpan<T>` companions in .NET 9 for exactly this reason; prefer them when authoring new variadic APIs.

First-class implicit span conversions (C# 14)

Before C# 14 you constantly wrote `.AsSpan()` to feed an array or string into a span-typed parameter, and — more painfully — generic type inference frequently failed across the conversion. C# 14 makes these conversions *first-class*: the language knows about them, so they participate in overload resolution and type inference like any built-in conversion. The recognized conversions are array `T[]` → `Span<T>`, array → `ReadOnlySpan<T>`, `Span<T>` → `ReadOnlySpan<T>`, `string` → `ReadOnlySpan<char>`, and a covariant `Derived[]` → `ReadOnlySpan<Base>`.

```

static void Consume(ReadOnlySpan<byte> bytes) { /* ...
    */ }

byte[] buffer = [1, 2, 3, 4];
Consume(buffer); // C# 14: array →
                  ReadOnlySpan<byte> implicitly. Pre-14:
                  Consume(buffer.AsSpan())

static int CountVowels(ReadOnlySpan<char> text)
{
    int n = 0;
    foreach (char c in text)
        if ("aeiou".AsSpan().IndexOf(char.ToLowerInvariant(c)) >= 0) n++;
    return n;
}

int v = CountVowels("automaton"); // C# 14: string →
                                   ReadOnlySpan<char> implicitly

```

The genuinely new capability is that the conversion flows through **generic inference**. Consider an array landing in a span-typed generic API:

```

// A generic span algorithm in the BCL style.
static T Last<T>(ReadOnlySpan<T> span) => span[^1];

string[] names = ["Ada", "Linus", "Grace"];

// C# 14: 'names' (string[]) implicitly converts to
//         ReadOnlySpan<string>,
// AND T is inferred as string from that conversion.
// Both were impossible pre-14.
string last = Last(names); // "Grace"

// Covariance also flows: string[] →
//         ReadOnlySpan<object>
static void Dump(ReadOnlySpan<object> items) { foreach
    (var i in items) Console.WriteLine(i); }
Dump(names); // string[] → ReadOnlySpan<object>
              implicitly (reference covariance)

```

Pre-C# 14 you had to write `Last(names.AsSpan())` and it still might not infer `T` because the conversion wasn't a language conversion the inference engine reasoned about. There is one tradeoff worth flagging: because `string` now binds to

`ReadOnlySpan<char>` overloads implicitly, adding a `ReadOnlySpan<char>` overload alongside an existing `string` overload can shift overload resolution. The compiler prefers the more specific match, but in genuinely ambiguous cases C# 13-compiled callers and C# 14-compiled callers can pick different overloads. When in doubt, keep the overload set unambiguous or use `[OverloadResolutionPriority]` (C# 13) to pin the intended winner.

Worked example: a zero-allocation tokenizer

This ties slicing, ranges, and `ReadOnlySpan<char>` together. We tokenize a CSV-ish line into fields without allocating a single string until the caller decides it actually needs one. The tokenizer is itself a `ref struct` because it *holds* a span — exactly the constraint the next section explains.

```

ref struct LineTokenizer
{
    private ReadOnlySpan<char> _remaining;
    private readonly char _separator;

    public LineTokenizer(ReadOnlySpan<char> line, char
        separator = ',')
    {
        _remaining = line;
        _separator = separator;
        Current = default;
    }

    public ReadOnlySpan<char> Current { get; private
        set; }

    // Enumerator pattern: foreach binds to
        GetEnumerator/MoveNext/Current by name.
    public LineTokenizer GetEnumerator() => this;

    public bool MoveNext()
    {
        // Once the final field (the one with no
            trailing separator) has been
            yielded, we're done. _consumedAll guards
            against re-yielding it.
        if (_consumedAll) return false;

        int idx = _remaining.IndexOf(_separator);
        if (idx < 0)
        {
            Current = _remaining.Trim(); // last
                field - no separator follows
            _remaining = default;
            _consumedAll = true;
            return true;
        }

        Current = _remaining[..idx].Trim(); // field
            up to the separator
        _remaining = _remaining[(idx + 1)..]; // skip
            past it - no allocation
        return true;
    }

    private bool _consumedAll;
}

// Usage - parse " alice , 42 , true " with zero
    string allocations:
ReadOnlySpan<char> line = " alice , 42 , true ";

```

```

foreach (ReadOnlySpan<char> field in new
    LineTokenizer(line))
{
    // field is a window into 'line'; materialize only
    // when needed.
    if (field.SequenceEqual("true"))
        Console.WriteLine("flag set");
    else if (int.TryParse(field, out int n)) //
        int.TryParse(ReadOnlySpan<char>, ...) - no
        substring
        Console.WriteLine($"number: {n}");
    else
        Console.WriteLine($"text:
            {field.ToString()}"); // ToString allocates -
            and only here
}
// Output:
// text: alice
// number: 42
// flag set

```

Every `_remaining[(idx+1)..]` and `[..idx]` is a two-field copy (reference + length); `Trim()` returns a sub-span, not a new string; `int.TryParse` and `SequenceEqual` accept spans directly. The only allocation in the whole loop is the explicit `field.ToString()` for the text field — and that is deliberate and visible. A `string.Split + Trim + int.Parse` version of this would allocate an array and three or four strings per line. Over a million-line file the span version's allocation profile is essentially flat.

ref struct : the stack-only guarantee

`Span<T>` contains a `ref` into someone else's memory. If you could put a `Span<int>` referring to `stackalloc int[5]` onto the GC heap — in a field of a class, in an array, captured by a closure — then after the creating method returned, that interior pointer would dangle into a reclaimed stack frame. The GC also cannot track or relocate an arbitrary interior pointer stored in a non-`ref` location. So the language forbids the situation categorically: any type with a `ref` field (or a field that is itself a `ref struct`) **must** be declared `ref struct`, and `ref struct` values are confined to the stack.

The rules the compiler enforces for a `ref struct` :

- It cannot be a field of a class or of a non- `ref` struct.
- It cannot be boxed: no assignment to `object` , `dynamic` , or — historically — an interface (relaxed by C# 13, see below).
- It cannot be an array element (`Span<int>[]` is illegal).
- It cannot be captured by a lambda or local function closure (closures lift captured locals to a heap-allocated display class).
- It cannot be a generic type argument **unless** the parameter says `allows ref struct` (C# 13).
- It cannot cross an `await` or `yield` boundary, because the async/iterator state machine is a generated class that lifts locals into heap fields.

That last one is the rule developers hit most:

```
async Task BrokenAsync(int[] data)
{
    Span<int> view = data.AsSpan(0, 4); // CS4012: a
                                     // ref struct local cannot be
    await Task.Delay(10);              // declared
                                     // in an async method at all -
    view[0] = 1;                       // the state
                                     // machine would have to heap-store it
}
```

The fix is *not* to suppress the error; it is to use `Memory<T>` , covered later.

`ref` fields, `scoped` , and the ref-safety model

A `ref struct` may declare `ref` fields (C# 11+), which is how you build your own span-like wrappers over a caller's buffer. The compiler's **ref-safety** analysis assigns each ref-bearing value a *safe-to-escape* scope and refuses any operation that would let a reference outlive the storage it points at. Two scopes matter: a value whose ref-lifetime is *the current method* (cannot be returned) versus one that is *the caller* (safe to return).

Default scoping depends on the parameter's type. A

`ref / in / out` parameter whose type is itself a `ref struct` is **implicitly scoped** (it may not escape via the return — a rule added so existing span-taking APIs stayed safe when `ref` fields landed in C# 11). A `ref` parameter to an ordinary type like `int` is **not** scoped: its reference is allowed to escape into a `ref` field or a `ref` return. The `scoped` modifier (C# 11) lets you *explicitly shorten* a reference's lifetime to “current method,” which is the opposite of what you need when you want to capture it. Here is a `ref struct` that stores a `ref` into a caller-supplied `int` — note the constructor parameter is **not scoped**, precisely because it must be allowed to escape into the field:


```

ref struct Accumulator
{
    private ref int _slot;           // ref field: points
                                    // into the CALLER's storage
    private int _count;

    // A plain (non-scoped) 'ref int' is escapable:
    // the compiler permits storing
    // it in the ref field, and in return BOUNDS the
    // Accumulator's safe-to-escape
    // scope to that of 'target'. Marking it 'scoped'
    // here would be a COMPILE ERROR
    // (CS9079) – a scoped ref may not be captured in
    // a ref field.
    public Accumulator(ref int target)
    {
        _slot = ref target;         // ref-field
                                    // assignment ('= ref'), binds the field
        _count = 0;
    }

    public void Add(int delta)
    {
        _slot += delta;             // mutate the
                                    // caller's variable through the ref
        _count++;
    }

    public readonly int Count => _count;
}

static void Demo()
{
    int running = 0;
    var acc = new Accumulator(ref running); // acc's
                                            // lifetime ≤ running's lifetime
    acc.Add(10);
    acc.Add(5);
    Console.WriteLine(running); // 15 – mutated
                                // through the ref field
    Console.WriteLine(acc.Count); // 2
}

```

Because `running` and `acc` are both locals in `Demo`, the ref field never outlives its target. If you tried to *return* `acc` from `Demo`, the compiler would reject it: `acc` captured a ref into `running`, which dies when `Demo` returns, so the compiler treats `acc`'s safe-to-escape scope as no wider than `running`'s. That bound-

ing is automatic *because* the parameter was escapable (non-scoped) — it is exactly what lets the constructor capture the reference while still keeping the result provably safe.

The mental model: **scoped** shrinks a reference's lifetime to the current method so it provably can't be captured or returned; the absence of **scoped** on a **ref** lets it flow out — into a **ref** field or a **ref** return — at the price of bounding any container's lifetime to the argument's.

Add **scoped** when you want to *promise* a borrow stays local (which also relaxes the caller's own escape analysis); leave it off (or apply **[UnscopedRef]** to a **ref struct**-typed parameter that is scoped by default) when you genuinely intend a reference to be captured or flow to the caller.

ref locals, **ref** returns, and returning interior references safely

A **ref** return hands the caller a live alias into existing storage. It is safe **only** when the referenced storage outlives the call — an array element, a field of a heap object, a **ref** / **scoped** parameter the caller owns — and **never** when it points at a local or **stackalloc** in the returning frame.

```
// SAFE: the array outlives the call; we return an
//      alias into it.
static ref int FindOrThrow(int[] array, int value)
{
    for (int i = 0; i < array.Length; i++)
        if (array[i] == value) return ref array[i];
    throw new KeyNotFoundException();
}

static void UseRefReturn()
{
    int[] xs = [1, 2, 3];
    ref int slot = ref FindOrThrow(xs, 2); // ref
    //      local aliasing xs[1]
    slot = 99;                             // writes
    //      through
    Console.WriteLine(xs[1]);               // 99
}
```

The dangling-reference mistake the compiler catches for you:

```
static ref int Dangling()
{
    int local = 42;
    return ref local; // CS8168: cannot return a ref
                      // to a local - 'local' dies here
}

static ref int DanglingStack()
{
    Span<int> tmp = stackalloc int[1];
    return ref tmp[0]; // CS8157 / escape error: stack
                      // memory freed on return
}
```

These are *compile errors*, not runtime faults — the ref-safety pass exists precisely so a dangling interior reference can never reach IL. The BCL leans on this hard: `List<T>` exposes `CollectionsMarshal.AsSpan(list)` to hand out a `Span<T>` over its internal array, and `ref` returns let `Dictionary`'s `CollectionsMarshal.GetValueRefOrNullRef` mutate a value in place with no copy.

allows ref struct and ref struct interfaces (C# 13)

For years you could not write a generic algorithm whose type parameter could be instantiated with `Span<T>`, because a `ref struct` couldn't be a type argument at all. C# 13's `allows ref struct` anti-constraint fixes this. It is the first *anti*-constraint in the language: instead of narrowing the allowed types, it *widens* them to admit `ref struct`s. The compiler then enforces ref-safety on the generic body as if `T` might be stack-only — so inside such a method you cannot box `T`, store it in a heap field, or let it escape across `await`.

Note the limitation that drives the design: `Span<T>` and `ReadOnlySpan<T>` do **not** implement `IEnumerable<T>` / `IReadOnlyList<T>` (those require boxing into an enumerator and are off-limits to a `ref struct`), so you cannot abstract over “array or span” through a collection interface. The lever is a constraint the value type *can* satisfy — most usefully a `ref struct` -

friendly interface of your own (next subsection), or no member constraint at all when the body only needs operations the JIT can resolve per type. A minimal example just admits the `ref struct` as a type argument and forwards it:

```
// 'allows ref struct' lets a Span<T> be the type
// argument T. With no member
// constraint the body can't index T directly, so this
// wrapper forwards the
// real work to a span-typed delegate – the point is
// that T may now be a ref struct.
static R Run<T, R>(T value, Func<T, R> body)
    where T : allows ref struct // anti-constraint
    must come last in the clause
    => body(value);

Span<int> nums = stackalloc int[] { 3, 1, 4, 1, 5 };
// Pre-C# 13 this call was impossible: Span<int> could
// not be a type argument.
int max = Run(nums, static s =>
{
    int m = int.MinValue;
    foreach (int x in s) if (x > m) m = x;
    return m;
});
Console.WriteLine(max); // 5
```

Here `R` is an ordinary type parameter (no anti-constraint), so the result can flow out normally; only `T` is widened to admit a stack-only span. The lambda's parameter `s` is typed `Span<int>` via inference, and because it is a `static` lambda it captures nothing — important, since a `ref struct` could never be captured anyway.

Pair it with **ref struct interface implementation** (also C# 13): a `ref struct` may now implement an interface, *but* it can never be converted to that interface type, because that conversion is a boxing operation and boxing a `ref struct` is forbidden. So the interface is usable only through a generic constraint, never as a variable type:

```

interface IByteSink
{
    void Write(byte b);
    int Written { get; }
}

// A ref struct implementing an interface, writing
// into a caller's span.
ref struct SpanSink : IByteSink
{
    private readonly Span<byte> _buffer;
    private int _pos;
    public SpanSink(Span<byte> buffer) { _buffer =
        buffer; _pos = 0; }
    public void Write(byte b) => _buffer[_pos++] = b;
    public int Written => _pos;
}

// Generic over the interface AND allowing ref structs
// - calls dispatch WITHOUT boxing.
static void WriteHeader<TSink>(ref TSink sink)
    where TSink : IByteSink, allows ref struct
{
    sink.Write(0xCA);
    sink.Write(0xFE);
}

static void UseSink()
{
    Span<byte> buf = stackalloc byte[8];
    var sink = new SpanSink(buf);
    WriteHeader(ref sink);           // constrained
    // call - no box, no virtual dispatch overhead
    Console.WriteLine(sink.Written); // 2
    Console.WriteLine(buf[0]);       // 202 (0xCA)

    // IByteSink box = sink;         // ERROR: cannot
    // box a ref struct to its interface
}

```

Two practical constraints to remember: `allows ref struct` cannot be combined with the `class` / `class?` constraint (a stack-only type is obviously never a reference type), and it must appear *last* in the constraint clause. Pass the value `ref` (or `in`) as shown — passing a large `ref struct` by value copies its fields, and passing it through the interface by value defeats the no-box guarantee’s intent for mutation.

`Memory<T>` and `ReadOnlyMemory<T>` : heap-storable, async-safe slices

`Memory<T>` exists to solve the one thing `Span<T>` structurally cannot do: **survive on the heap and across `await`**.

`Memory<T>` is an ordinary (non-`ref`) struct, so it can be a class field, a generic type argument, captured in a closure, and stored in an async state machine. It does *not* contain a raw interior pointer; it holds an object reference (the array, a `MemoryManager<T>`, or a string), an offset, and a length, and it materializes a `Span<T>` on demand via the `.Span` property:

```
async Task FixedAsync(byte[] data)
{
    Memory<byte> region = data.AsMemory(0, 4); // OK:
    Memory<T> is heap-safe
    await Task.Delay(10); //
    survives the await
    region.Span[0] = 1; //
    re-project to a Span only inside a sync region
}
```

The discipline: **pass and store `Memory<T>` ; convert to `Span<T>` only inside a synchronous block, and never let that `Span` straddle an `await`**. The `.Span` projection is cheap but not free (it validates and reconstructs the span), so cache it in a local for tight inner loops *within* a sync section, then discard it before the next `await`.

Converting an async method from `Span` to `Memory`

This is the single most common refactor in this space. The broken version captures a `Span<T>` across `await` and will not compile:

```

// BROKEN – does not compile.
async Task<int> SumChunkedAsync_Broken(Stream stream,
    int[] backing)
{
    Span<int> view = backing.AsSpan();    // CS4012:
    ref struct local in an async method
    int total = 0;
    while (true)
    {
        int read = await ReadIntsAsync(stream, view);
        // ...and it would have to live across await
        if (read == 0) break;
        foreach (int x in view[..read]) total += x;
        view = view[read..];
    }
    return total;
}

```

The fix carries `Memory<T>` across the boundary and projects to `Span<T>` only in the synchronous accumulation loop:

```
// FIXED – Memory<T> crosses the await; Span<T> is
// confined to a sync block.
async Task<int> SumChunkedAsync(Stream stream, int[]
    backing)
{
    Memory<int> view = backing.AsMemory(); // heap-
    // safe; legal across await
    int total = 0;
    while (true)
    {
        int read = await ReadIntsAsync(stream, view);
        // pass Memory<int>, not Span<int>
        if (read == 0) break;

        // Project to Span ONLY here, inside a
        // synchronous region – never spanning an await.
        ReadOnlySpan<int> chunk = view.Span[..read];
        foreach (int x in chunk) total += x;

        view = view[read..]; // Memory's range slicing
        // also allocation-free
    }
    return total;
}

// The async I/O primitive takes Memory<T>, mirroring
// Stream.ReadAsync(Memory<byte>, ...).
static ValueTask<int> ReadIntsAsync(Stream stream,
    Memory<int> destination)
=> /* read into
    MemoryMarshal.AsBytes(destination.Span) ... */
    ValueTask.FromResult(0);
```

This mirrors the BCL exactly: `Stream.ReadAsync` and `PipeReader` take `Memory<byte>` / `ReadOnlyMemory<byte>`, never `Span<byte>`, because they `await`. Whenever you see an async signature wanting a buffer, it must be `Memory<T>`; whenever you see a synchronous hot-path signature, prefer `ReadOnlySpan<T>`.

Performance notes, IL/JIT, and the GC

- **No allocation, but not free of cost.** Constructing a `Span<T>` copies two fields; the JIT keeps them in registers in tight loops. The indexer's bounds check is real, but the JIT routinely *elides* it when it can prove the index is in range

(e.g. a `for (i = 0; i < span.Length; i++)` loop hoists and removes the per-iteration check). Writing the loop against `span.Length` rather than a separately captured length is what lets the JIT see the invariant.

- **`stackalloc` is the partner of `Span<T>` , with a ceiling.** Stack-allocated buffers have zero GC impact and perfect locality, but the stack is ~1 MB by default. The standard pattern is a stack buffer for the common small case with a heap fallback (or `ArrayPool<T>`) above a threshold:

```
const int StackThreshold = 256;
Span<byte> buffer = length <= StackThreshold
    ? stackalloc byte[StackThreshold] // hot
      path, no allocation
    : new byte[length];               // cold
      path, heap (or ArrayPool.Rent)
buffer = buffer[..length];
```

- **`Memory<T>.Span` has overhead `Span<T>` doesn't.** It re-validates and reconstructs the span from the stored object+offset+length on every access. In an inner loop, project once into a local; don't call `.Span` per iteration.
- **`ArrayPool<T>.Shared` is the right backing for large, reusable buffers** — rent, wrap in a `Span<T>` / `Memory<T>` , `return` in a `finally` . It turns repeated large allocations into amortized reuse and keeps the Gen-2 heap quiet.
- **Generic span code via `allows ref struct` dispatches without boxing** — the constrained call is monomorphized by the JIT per value-type `T` , so there is no virtual-dispatch or boxing cost even though an interface is in the signature.

Pitfalls checklist

- **Storing a `Span<T>` where it can't live.** A `Span<T>` field on a class, an element of `Span<int>[]` , capture in a lambda, or a local that survives `await` / `yield` are all compile errors — the fix is `Memory<T>` , never a suppression. The error is the language doing its job.

- **Returning a ref into the current frame.** `return ref local;` and `return ref stackalloc'd[0];` dangle and are rejected (CS8168 / escape errors). Only return refs into storage that outlives the call — arrays, heap fields, caller-owned `ref` / `scoped` parameters.
- **Off-by-one with `^` and over-slicing.** `^0 == Length` (one past the end), so `span[^0]` throws; the last element is `span[^1]`. `Slice` / range over-runs (`span[1..4]` on a length-3 span) throw `ArgumentOutOfRangeException` — they fail loudly rather than read out of bounds, but they still fail.
- **Forgetting that `Memory<T>.Span` must not straddle `await`.** It's legal to *hold* `Memory<T>` across `await`; it's illegal to hold the `Span<T>` you projected from it. Project inside the synchronous region only.
- **Overload ambiguity from C# 14 implicit `string` → `ReadOnlySpan<char>`.** Adding a span overload next to a string overload can change which one a call binds to. Keep overload sets unambiguous or pin intent with `[OverloadResolutionPriority]` (C# 13).

When *not* to reach for spans

Spans are a precision instrument for hot, allocation-sensitive, synchronous paths — parsers, serializers, formatters, buffer plumbing. They are the wrong tool when the ref-struct constraints cost more than they buy: data that is inherently long-lived or shared across `await` boundaries (use `Memory<T>`, or just arrays), public APIs whose callers think in `IEnumerable<T>` / `List<T>`, or code that isn't on a measured hot path. A `Span<T>` that forces you to restructure a method into a synchronous helper just to satisfy the ref-safety analyzer, with no measured allocation win, is premature optimization wearing a performance costume. Reach for `ReadOnlySpan<T>` first in *new* low-level parsing code, reach for `Memory<T>` the moment an `await` enters the picture, and profile before you re-shape an existing API around either.

Sources for version-specific feature confirmation: [What's new in C# 14 — Microsoft Learn](#), [First-class span types — C# feature specification](#), [What's new in C# 13 — Microsoft Learn](#), [Allow ref struct types to implement interfaces — C# 13 spec](#), [ref struct types — C# reference](#), [Constraints on type parameters — C#](#).

14 · `stackalloc`, Inline Arrays, and Fixed-Size Buffers

The fastest allocation is the one that never touches the GC heap. This chapter is about three mechanisms that let you keep small, short-lived, fixed-size buffers on the stack or inline inside a value type — `stackalloc`, the `[InlineArray]` attribute (C# 12), and their interaction with `Span<T>` — and about the precise rules and footguns that separate a 10× throughput win from a `StackOverflowException` you cannot catch.

The shape of the problem

A surprising amount of hot-path code allocates a tiny array, fills it, reads it, and throws it away within a single method call: a 16-byte hash, a UTF-8 encoding scratch buffer, a list of three integers passed to a formatting routine, a key for a dictionary lookup. Each of those `new byte[16]` or `new int[3]` is a heap allocation that the GC must later collect. Individually cheap; in aggregate, across millions of calls, they are the dominant cost — Geno collections, cache pressure, and allocation-rate-driven latency spikes.

The tools in this chapter remove that allocation entirely. `stackalloc` carves the buffer out of the current frame's stack. `[InlineArray]` embeds the buffer's storage directly inside a struct's layout. Both produce something you consume through `Span<T>`, which is the unifying abstraction: a length-checked window over contiguous memory regardless of where that memory lives. The danger is symmetric — stack memory and inline-struct memory both have *lifetimes tied to a frame or an owner*, and `Span<T>`'s `ref struct` rules exist precisely to stop you from outliving them.

`stackalloc` semantics

`stackalloc T[n]` allocates `n * sizeof(T)` bytes from the current method's stack frame. `T` must be an unmanaged type

(no managed references), which is why you `stackalloc byte`, `int`, `char`, or your own `struct` of unmanaged fields, but never `stackalloc string[]`. Whether the memory is zeroed is governed by the method's `localsinit` flag, not by whether you assign to a pointer or to a `Span<T>`: the C# spec leaves `stackalloc` contents *undefined*, but the compiler emits `localsinit` by default, which forces the JIT to zero the entire local stack region — so in practice both the pointer form and the span form start as all zeros unless the enclosing method (or module/type) is annotated with `[SkipLocalsInit]`, which suppresses that flag and hands you uninitialized stack memory. Do not rely on the zeroing as a language guarantee; treat it as “zeroed unless `[SkipLocalsInit]` is in scope.”

Since C# 7.2 the idiomatic form assigns to a span — no `unsafe` required:

```
Span<byte> buffer = stackalloc byte[256];
buffer[0] = 0xFF;
// buffer is valid only until this method returns.
```

The lifetime is the *frame*, not a lexical block. A `stackalloc` inside an `if` block is not freed when the block exits — it lives until the method returns. (This matters for the loop pitfall below.) The `Span<byte>` you get is a `ref struct`: it cannot be boxed, stored in a field of a normal class, captured by a lambda, or used across an `await` / `yield`. The compiler's escape analysis enforces that the span never leaves the frame that owns the stack memory.

Three more semantic points worth internalizing:

- **No `free`**. Stack memory is reclaimed by the frame pointer moving on return. There is no per-buffer cleanup, which is exactly why it is fast — and why you cannot “release early.”
- **It counts against the stack budget.** The default managed thread stack is 1 MB on Windows and typically 8 MB on Linux for the main thread but **1 MB for thread-pool and explicitly created threads** unless you override it. A

`stackalloc byte[1_000_000]` is a near-guaranteed crash on a pool thread.

- **Implicitly-typed `stackalloc[]`** (C# 7.3) and seeding from a collection expression (C# 12) let you write

`stackalloc[] { 1, 2, 3 }` or `Span<int> s = [1, 2, 3]` — more on that later.

The stack budget, and why overruns are uncatchable

A `StackOverflowException` is not a normal exception. Since .NET Framework 2.0 and continuing through .NET 10, the runtime treats it as **non-recoverable**: it cannot be caught, it bypasses `finally` blocks and `catch (Exception)`, and it terminates the process. There is no `try / catch` that saves you. This is the single most important fact in the chapter, because it changes how you must reason about `stackalloc` sizes: an oversized `stackalloc` is not “an error you handle,” it is “a process you crash.”

The corollary: **the size of a `stackalloc` must be bounded by a compile-time-known constant, never by untrusted input**. If a length comes from a request header, a file, or a user, it must pass through a threshold guard before it is allowed anywhere near the stack.

Guarding `stackalloc` with a threshold and the array pool

The canonical pattern pairs a small stack buffer for the common case with `ArrayPool<T>` for the overflow case, unifying both behind a single `Span<T>`:

```

using System.Buffers;

static int CountUtf8Bytes(ReadOnlySpan<char> text)
{
    const int StackThreshold = 256;

    int maxBytes =
        System.Text.Encoding.UTF8.GetMaxByteCount(text
            .Length);

    // Rent from the pool above the threshold;
    stackalloc below it.
    byte[]? rented = maxBytes > StackThreshold
        ? ArrayPool<byte>.Shared.Rent(maxBytes)
        : null;

    Span<byte> buffer = rented is null
        ? stackalloc byte[StackThreshold]
        : rented;

    try
    {
        int written =
            System.Text.Encoding.UTF8.GetBytes(text,
                buffer);
        return written; // do real work on
        buffer[..written] here
    }
    finally
    {
        if (rented is not null)
            ArrayPool<byte>.Shared.Return(rented);
    }
}

```

Several details make this correct rather than merely plausible:

- **The threshold is the `stackalloc` size, not the data size.** We always `stackalloc StackThreshold` (a constant), never `stackalloc maxBytes`. That keeps the stack draw fixed and known.
- **The `rented / null` split picks the buffer source exactly once.** `Span<byte> buffer = rented is null ? stackalloc ... : rented;` is legal because the explicit `Span<byte>` target type drives a *target-typed conditional expression* (C# 9): the `stackalloc` arm is already a

`Span<byte>` , and the `byte[]` arm converts to `Span<byte>` through `Span<T>`’s built-in array conversion. Without the explicit target type — e.g. with `var` — this fails with **CS0173** (“no implicit conversion between the two arms”), because `stackalloc byte[...]` and `byte[]` have no common type on their own. (C# 14’s *first-class span conversions* add new implicit array→`Span` / `ReadOnlySpan` , `string` → `ReadOnlySpan<char>` , and covariant `Span` → `ReadOnlySpan` conversions that participate in overload resolution and type inference, but they are not what makes *this* target-typed conditional compile — that has worked since C# 9.)

- **Rent may return a larger array than requested.**
Always slice to the length you actually wrote (`buffer[..written]`); never assume `buffer.Length == maxBytes` .
- **Return is in a finally** . A leaked rental is not a crash, but it defeats the pool. Do not `Return` in the `stackalloc` arm — there is nothing to return, and `rented` is `null` .

One subtlety the compiler’s escape analysis gets right: assigning a `stackalloc` result to a `Span<byte>` that *also* might hold a pooled array is fine, because the span’s escape scope is the narrower of the two. You cannot `return buffer;` from this method — the compiler rejects it because *one* of the possible backing stores is stack memory.

When you want disposal symmetry rather than a hand-rolled `try / finally` at every call site, a thin `ref struct` owner is the standard idiom for “stackalloc-or-pool”:


```

ref struct ScratchBuffer<T> where T : unmanaged
{
    private T[]? _rented;
    public Span<T> Span { get; }

    public ScratchBuffer(Span<T> stackSpace, int
        needed)
    {
        if (needed <= stackSpace.Length)
        {
            Span = stackSpace[..needed];
            _rented = null;
        }
        else
        {
            _rented =
                ArrayPool<T>.Shared.Rent(needed);
            Span = _rented.AsSpan(0, needed);
        }
    }

    public void Dispose()
    {
        if (_rented is not null)
            ArrayPool<T>.Shared.Return(_rented);
    }
}

// Usage – the stackalloc lives in the CALLER's frame,
// which is required:
static void Process(int count)
{
    using var scratch = new ScratchBuffer<int>
        (stackalloc int[128], count);
    Span<int> work = scratch.Span;
    // ... fill and use work ...
}

```

The reason `stackalloc int[128]` is passed in by the caller and not allocated inside the constructor is fundamental: a `stackalloc` inside `ScratchBuffer`'s constructor would be freed when the constructor returns. Because `ScratchBuffer` is a `ref struct` and `Span<T>` is a `ref struct` field of it, the escape rules force the stack buffer to be owned by the frame that uses it. This is the one ergonomic wart of the pattern, and it is load-bearing — do not try to “fix” it by allocating inside.

stackalloc inside a loop: the accumulation trap

`stackalloc`'s frame lifetime means a `stackalloc` in a loop body does **not** reclaim memory each iteration. The stack draw accumulates for the life of the method:

```
// WRONG: each iteration adds 1 KiB to the frame;
//          10_000 iterations ≈ 10 MiB → crash.
static void Bad(int iterations)
{
    for (int i = 0; i < iterations; i++)
    {
        Span<byte> tmp = stackalloc byte[1024]; //
        // accumulates!
        Work(tmp);
    }
}

// RIGHT: allocate once outside the loop, reuse and
//         clear as needed.
static void Good(int iterations)
{
    Span<byte> tmp = stackalloc byte[1024];
    for (int i = 0; i < iterations; i++)
    {
        tmp.Clear(); // if you need a fresh zeroed
        // buffer
        Work(tmp);
    }
}
```

The accumulation is the real, observable behavior — `localloc` grows the frame on every execution and nothing reclaims it until the method returns — which is exactly why the .NET SDK ships analyzer rule **CA2014 (“Do not use stackalloc in loops”)**, enabled by default since .NET 5 at *warning* severity (worth elevating to error with `dotnet_diagnostic.CA2014.severity = error`). (Recent JIT work in .NET 10 can reuse the stack slot for a repeated constant-size `localloc` at a single site, but you must not depend on it: it does not apply to variable sizes, and any refactor that changes the allocation site reintroduces the accumulation.) The rule is mechanical and non-negotiable: **never put `stackalloc` in a loop body**. Allocate above the loop, `Clear()` inside if you need zeroing. Let CA2014 catch it for you.

Inline arrays: fixed-size value-type buffers

Introduced in C# 12 (.NET 8), the `[InlineArray]` attribute turns a struct with a *single* field into a fixed-length, contiguous-laid-out buffer whose storage lives wherever the struct lives — on the stack as a local, inline inside another struct, or inline inside a heap object's fields. No separate array object, no extra indirection, no GC reference to chase.

```
using System.Runtime.CompilerServices;

[InlineArray(8)]
public struct Buffer8<T>
{
    private T _element0; // exactly one field; the
                        // attribute replicates it 8x
}
```

The runtime (new in .NET 8) expands `_element0` into 8 contiguous `T` slots. The struct's `sizeof` is `8 * sizeof(T)` (plus alignment), and its memory layout matches `T[8]`'s *element sequence* — but without the array header. The compiler then grants this type three superpowers it gives to no ordinary struct:

```
var buf = new Buffer8<int>();

buf[0] = 10;           // indexer synthesized by the
                       // compiler
buf[7] = 70;           // buf[8] would be a COMPILE-
                       // TIME error (constant out of range)
int x = buf[3];

Span<int> span = buf;   // implicit
                       // conversion to Span<int> (whole buffer)
ReadOnlySpan<int> ro = buf; // and to
                       // ReadOnlySpan<int>
Span<int> slice = buf[2..5]; // slicing →
                       // Span<int> of length 3

foreach (int v in buf) { /* ... */ } // enumeration
                                     // via the span
```

The indexing, slicing, and span conversion are **compiler-synthesized directly against the buffer's memory** — no

`MemoryMarshal` call appears in your source, and for constant indices the compiler even elides bounds checks and can report out-of-range constants at compile time. Under the hood, the conversion to `Span<T>` is equivalent to

`MemoryMarshal.CreateSpan(ref Unsafe.As<Buffer8<T>, T>(ref buf), 8)`, which is exactly what you write by hand when you need an explicit `AsSpan()` method (for example, to expose a span from a `readonly` field or to return a slice with a method name):

```
[InlineArray(16)]
public struct Buffer16
{
    private byte _e0;

    public readonly ReadOnlySpan<byte>
        AsReadOnlySpan() =>
        MemoryMarshal.CreateReadOnlySpan(
            in
            System.Runtime.CompilerServices.Unsafe.AsRef(in
            _e0), 16);
}
```

The lifetime trap with inline arrays

Because an inline array's storage is *the struct itself*, the span you get from it lives exactly as long as the struct does. Treat the span like any other `ref` into the struct: if the struct is a stack local, the span must not escape the frame; if the struct is a field of a `ref struct`, the same escape rules cascade.

```
// WRONG - returns a span over a local that's about to
// die.
static Span<int> Leak()
{
    var buf = new Buffer8<int>();
    return buf;           // span over a local struct
                          // → compile-time error (good!)
}
```

The compiler catches the *direct* return because `Buffer8<int> → Span<int>` is treated like taking a `ref` to a local. The version it cannot always catch is the indirect one: stashing the span in a

`ref struct` field whose lifetime you've reasoned about incorrectly, or — the most common mistake — *mentally* treating the inline array as if it were a heap `int[]` whose span stays valid after the owning method returns. It does not. An inline array is value semantics all the way down: copy the struct, and you copy all the elements; let the struct die, and the span dangles.

Inline arrays vs `unsafe fixed` buffers vs arrays

Three ways to model “N elements inline.” Choose deliberately:

Property	<code>[InlineArray]</code> (C# 12)	<code>unsafe fixed T[n]</code>	<code>T[]</code> array
Element type	any valid type argument (managed <code>T</code> allowed; not a pointer or <code>ref struct</code>)	only the ~13 primitive blit-table types (<code>bool</code> , <code>byte</code> , <code>char</code> , <code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , and unsigned/ <code>sbyte</code> variants)	any <code>T</code>
<code>unsafe</code> required	no	yes (to declare and to index)	no
Bounds checking	yes (span/indexer)	no (raw pointer arithmetic)	yes
Storage location	inline in the owner	inline in the owner	separate heap object + header
Span access	implicit, safe	manual via pointer	<code>.AsSpan()</code>
GC heap allocation	none (when the owner is on the stack)	none	always
Generic over <code>T</code>	yes	no	yes

For contrast, the `unsafe fixed` form for the same 16-byte buffer looks like this — note the `unsafe` on both the declaration and every access:

```

unsafe struct LegacyBuffer16
{
    public fixed byte Data[16];    // only primitive
                                   blittable element types

    public byte Get(int i)
    {
        fixed (byte* p = Data)    // must pin to index
                                   – no bounds check
        {
            return p[i];           // p[16] is
                                   undefined behavior, not an exception
        }
    }
}

```

`unsafe fixed` buffers predate inline arrays and survive mainly in legacy interop code; they are restricted to a handful of primitive types, require `unsafe` at every use site, and give you no bounds checking. **For new code, `[InlineArray]` supersedes `fixed` buffers in essentially every scenario** — it is safe, generic, and span-friendly. Reach for `fixed` only when matching an existing `unsafe` struct definition you cannot change.

Versus a plain `T[]`: the array is heap-allocated and reference-typed, so it always costs an allocation and an indirection, but it can be freely returned, stored in fields, and shared. Use an inline array when the buffer is small, fixed-size, and *owned* (no sharing), and when avoiding the allocation matters. Use an array when the buffer is large, variable-size, long-lived, or shared.

Small-buffer optimization with inline arrays

The headline use case for `[InlineArray]` is *small-buffer optimization* (SBO): a collection that stores its first few elements inline and only spills to the heap when it grows past the inline capacity. Most instances never spill, so most instances never allocate. Here is a small-string builder that keeps short strings entirely on the stack/inline and falls back to `ArrayPool<char>` for longer ones:

```

using System.Buffers;
using System.Runtime.CompilerServices;

[InlineArray(32)]
struct InlineChars32
{
    private char _e0;
}

ref struct SmallStringBuilder
{
    private InlineChars32 _inline; // 32 chars
                                   stored inline, zero allocation
    private char[]? _rented; // overflow
                              storage from the pool
    private Span<char> _chars; // current
                               writable window (inline or rented)
    private int _pos;

    public SmallStringBuilder()
    {
        _rented = null;
        // 'this' is a ref struct local; the inline
        buffer lives in this frame.
        _chars = _inline; // implicit
                           InlineChars32 → Span<char>
        _pos = 0;
    }

    public int Length => _pos;

    public void Append(ReadOnlySpan<char> value)
    {
        if (_pos + value.Length > _chars.Length)
            Grow(_pos + value.Length);

        value.CopyTo(_chars[_pos..]);
        _pos += value.Length;
    }

    public void Append(char c)
    {
        if (_pos >= _chars.Length)
            Grow(_pos + 1);
        _chars[_pos++] = c;
    }

    private void Grow(int needed)
    {

```



```

    int newCap = Math.Max(needed, _chars.Length *
2);
    char[] next =
    ArrayPool<char>.Shared.Rent(newCap);

    _chars[.._pos].CopyTo(next);    // copy what
we have so far

    if (_rented is not null)
        ArrayPool<char>.Shared.Return(_rented);

    _rented = next;
    _chars = next;                // future
writes go to the rented array
}

public override readonly string ToString() =>
    new(_chars[.._pos]);

public void Dispose()
{
    char[]? toReturn = _rented;
    this = default;                //
defensively poison further use
    if (toReturn is not null)
        ArrayPool<char>.Shared.Return(toReturn);
}
}

```

Usage and the critical lifetime contract:

```

static string FormatCoordinate(double lat, double lon)
{
    var sb = new SmallStringBuilder();    // no heap
                                         // allocation yet
    try
    {
        sb.Append('(');
        Span<char> num = stackalloc char[32];
        lat.TryFormat(num, out int n1);
        sb.Append(num[..n1]);
        sb.Append(", ");
        lon.TryFormat(num, out int n2);
        sb.Append(num[..n2]);
        sb.Append(')');
        return sb.ToString();             //
                                         // materializes a heap string at the boundary
    }
    finally
    {
        sb.Dispose();
    }
}
// "(50.0875, 14.4214)" with zero intermediate heap
// allocations
// until the final string is produced.

```

The design notes that matter:

- **SmallStringBuilder is a ref struct**. It must be, because it holds a `Span<char>` field (`_chars`) that may point at inline storage. That constraint forbids it from being a field of a class, captured by a lambda, or living across an `await` — which is correct: its inline buffer is frame-bound.
- **`_chars = _inline`; aliases the inline buffer**. Once `Grow` runs, `_chars` is repointed at the rented array; the inline buffer is abandoned (its old contents already copied). Reads and writes go through `_chars` uniformly, so the inline-vs-heap distinction vanishes from the append logic.
- **The common case allocates nothing**. Strings up to 32 chars never touch the pool. `ToString()` is the only heap allocation, and it is unavoidable — you asked for a `string`.
- **`Dispose` returns the rental and self-poisons**. Forgetting `Dispose` leaks a pooled array (a slow leak, not a

crash). The `using / finally` discipline is the price of the optimization.

This pattern is exactly how `System.Text.ValueStringBuilder` (used internally throughout the BCL) and many SIMD/parsing hot paths are built. The BCL's own `Span<T>`-based builders pre-date `[InlineArray]` and use a caller-supplied `stackalloc` span instead of an inline field; the inline-array version above is the modern, self-contained equivalent.

Modeling native fixed-size struct fields for P/Invoke

Native structs routinely contain fixed-size arrays — `char Name[256]`, `byte Address[16]`, `uint Reserved[4]`. Before C# 12 you modeled these with `unsafe fixed` buffers (limited types, `unsafe` everywhere) or by expanding the array into N individual fields by hand. `[InlineArray]` models them cleanly, safely, and with correct blittable layout:

```

using System.Net;
using System.Net.Sockets;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Native:
// struct in6_addr { uint8_t s6_addr[16]; };
[InlineArray(16)]
struct In6Addr
{
    private byte _b0;
}

[StructLayout(LayoutKind.Sequential)]
struct SockAddrIn6
{
    public ushort sin6_family;    // AF_INET6
    public ushort sin6_port;      // network byte
    order
    public uint sin6_flowinfo;
    public In6Addr sin6_addr;     // 16 bytes,
    inline, no pointer
    public uint sin6_scope_id;
}

static class Native
{
    static SockAddrIn6 FromIPv6(IPAddress address,
        ushort port)
    {
        var sa = new SockAddrIn6
        {
            sin6_family =
                (ushort)AddressFamily.InterNetworkV6,
            sin6_port =
                (ushort)IPAddress.HostToNetworkOrder((short)port),
        };

        // Copy the 16 address bytes straight into the
        inline buffer.
        Span<byte> dst = sa.sin6_addr; //
        In6Addr → Span<byte>
        if (!address.TryWriteBytes(dst, out int
            written) || written != 16)
        {
            throw new ArgumentException("Not an IPv6
                address.", nameof(address));
        }

        return sa;
    }
}

```

```
}  
}
```

The win is layout fidelity *and* safety: `SocketAddrIn6` is blittable (all fields are unmanaged, the inline array contributes exactly 16 contiguous bytes), so it marshals to the native `sockaddr_in6` with `LayoutKind.Sequential` and no marshaller copy. You can `Marshal.SizeOf<SocketAddrIn6>()` it, pass it by `ref` to a P/Invoke, or write it through a `Span<byte>` of the whole struct — all without a single `unsafe` block. Compare to the old `fixed byte sin6_addr[16];`, which forced the entire enclosing struct to be `unsafe` and gave you a raw `byte*` to index without bounds checks.

A caution specific to interop: an inline array's element type must be unmanaged for the struct to remain blittable.

`[InlineArray(256)] struct CharBuf { private char _e0; }` is fine (`char` is blittable as UTF-16); an inline array of a managed type is legal C# but makes the struct non-blittable and unusable for direct P/Invoke.

Combining `stackalloc` with collection expressions

Collection expressions (C# 12) give `stackalloc` a tidy literal syntax for tiny, known-at-compile-time seeds. A target-typed collection expression assigned to a `Span<T>` or `ReadOnlySpan<T>` may be realized as a `stackalloc` by the compiler when the element count is small and the values are simple — you get the ergonomics of a literal with stack (or even inline `[InlineArray]`-buffer) storage and no heap array:

```
// C# 12 collection expression targeting a span.
ReadOnlySpan<int> primes = [2, 3, 5, 7, 11, 13];
// The compiler may emit a ReadOnlySpan over
//   RVA/static data (for constants)
// or a stackalloc – either way, no heap int[] is
//   allocated.

// Explicit stackalloc with an initializer (C# 7.3+,
//   still valid):
Span<int> deltas = stackalloc int[] { -1, 0, 1 };

// Implicitly-typed stackalloc initializer:
Span<int> small = stackalloc[] { 4, 8, 15, 16, 23, 42
    };
```

For a *truly* ephemeral tiny buffer seeded from variables, the collection-expression-into-span form reads cleanly:

```
static double Median3(double a, double b, double c)
{
    Span<double> v = [a, b, c];    // tiny, ephemeral;
    // no heap array
    v.Sort();
    return v[1];
}
```

Two caveats. First, for *constant* element lists targeting `ReadOnlySpan<T>`, the compiler prefers emitting a span over a static read-only data blob (zero stack, zero heap) — which is even better than `stackalloc`; you cannot mutate that span, which is why the example above uses `Span<double>` (mutable) and the prime list uses `ReadOnlySpan<int>` (constant). Second, do not lean on collection expressions for large or variable-length seeds expecting stack storage — past a compiler-internal size, or when the length isn't a compile-time constant, the target-typed `Span<T>` may be backed by a heap array instead. The size guarantees of `stackalloc` only come from writing `stackalloc` (or an `[InlineArray]` field) explicitly.

A worked example: SBO set for hot-path dedup

Tying the threads together — `stackalloc`, `[InlineArray]`, the pool fallback, and span-based work — here is a tiny set tuned for

the overwhelmingly common case of “at most a handful of distinct ints,” used in a hot loop where the typical input is small but the worst case must stay correct:

```

using System.Buffers;
using System.Runtime.CompilerServices;

[InlineArray(8)]
struct Inline8Int
{
    private int _e0;
}

ref struct TinyIntSet
{
    private Inline8Int _inline;
    private int[]? _rented;
    private Span<int> _items;
    private int _count;

    public TinyIntSet()
    {
        _rented = null;
        _items = _inline;    // 8 ints inline; zero
                             // allocation for ≤ 8 distinct values
        _count = 0;
    }

    // Returns true if 'value' was newly added; false
    // if already present.
    public bool Add(int value)
    {
        for (int i = 0; i < _count; i++)
            if (_items[i] == value)
                return false;

        if (_count == _items.Length)
            Grow();

        _items[_count++] = value;
        return true;
    }

    private void Grow()
    {
        int[] next =
            ArrayPool<int>.Shared.Rent(_items.Length * 2);
        _items[.._count].CopyTo(next);
        if (_rented is not null)
            ArrayPool<int>.Shared.Return(_rented);
        _rented = next;
        _items = next;
    }
}

```



```

public readonly int Count => _count;

public void Dispose()
{
    int[]? toReturn = _rented;
    this = default;
    if (toReturn is not null)
        ArrayPool<int>.Shared.Return(toReturn);
}

static int CountDistinct(ReadOnlySpan<int> input)
{
    var set = new TinyIntSet();
    try
    {
        foreach (int x in input)
            set.Add(x);
        return set.Count;
    }
    finally
    {
        set.Dispose();
    }
}

```

For inputs with ≤ 8 distinct values — the design assumption — `CountDistinct` performs **zero heap allocations**: the `Inline8Int` buffer lives inside `set`, which lives in `CountDistinct`'s frame. Linear scan beats a `HashSet<int>` for these tiny sizes (no hashing, no buckets, cache-resident contiguous storage), and the JIT keeps the whole thing in registers and stack slots. Only when distinct values exceed 8 does it rent from the pool, and even then it never allocates a fresh array on the GC heap. This is the same architectural pattern the BCL uses for `ValueListBuilder<T>` and small-collection fast paths through-out `System.Text.Json`, `Regex`, and the formatting stack.

Performance notes, IL, and JIT implications

- `stackalloc` is a single IL instruction (`localloc`) plus a zeroing memset (for the span form). The JIT lowers it to a frame-pointer adjustment; there is no allocator call, no GC

bookkeeping, nothing to free. For constant sizes the cost is essentially free.

- **[InlineArray] access compiles to direct field/offset arithmetic.** Indexing `buf[i]` becomes a bounds-checked load at `&buf + i*sizeof(T)`; the span conversion is a `MemoryMarshal.CreateSpan` that the JIT inlines to a pointer-and-length. No array header, no covariance check (a hidden cost of `T[]` element writes for reference types), and no second cache line for the array object.
- **Zeroing is not free for large buffers.** A `stackalloc byte[8192]` pays an 8 KB memset on every call. If you re-fill the whole buffer before reading it, that zeroing is wasted; for very hot, very large buffers consider a pooled array you `Clear()` only when needed, or measure whether `[SkipLocalsInit]` (an attribute that suppresses `localsinit`) is worth the sharp edge of reading uninitialized stack memory. `[SkipLocalsInit]` is an expert tool — it makes `stackalloc` return *garbage*, not zeros, and is only safe when you provably write before you read.
- **Inlining and escape analysis interact.** A `ref struct` SBO type that gets fully inlined often has its inline buffer promoted into the caller's registers/stack entirely; the abstraction has zero runtime overhead. But the same `ref struct` rules that enable this forbid the type from crossing async/iterator boundaries — by design.

When NOT to use these features

- **Large or unbounded buffers.** Anything whose size derives from input, or exceeds a few hundred bytes to a couple of KB, belongs in `ArrayPool<T>` or a plain array — never raw `stackalloc`.
- **Long-lived or shared buffers.** If the buffer outlives the method or is handed to another component, it cannot be a `stackalloc` span or an inline-array span. Use an array (pooled or not).

- **Anything that must cross `await` / `yield`.** `Span<T>` and `ref struct` cannot live across these suspension points. C# 13 relaxed `ref` locals and `unsafe` blocks *within* iterators and async methods, but a `Span<T>` still cannot survive the `await` itself. Buffer your data into an array before awaiting.
- **Public API surface.** Exposing a `Span<T>` that aliases an inline-array field, or returning a `stackalloc` span, is impossible by construction — and exposing a `ref struct` SBO type forces every consumer into the same lifetime constraints. Keep these types `internal` / `private` implementation details; expose `string`, arrays, or `IReadOnlyList<T>` at boundaries.

The three pitfalls, restated as rules

1. **Never size a `stackalloc` from untrusted input.** A user-driven length is a `StackOverflowException` waiting to happen, and that exception **cannot be caught** — it kills the process, skips `finally`, and is indistinguishable from a crash. Always route variable lengths through a constant-bounded threshold guard with an `ArrayPool` fallback.
2. **Never `stackalloc` inside a loop.** Stack memory is reclaimed at frame exit, not block exit, so a loop accumulates stack draw every iteration. Allocate one buffer above the loop and `Clear()` it inside if you need a fresh zeroed view.
3. **Never let an inline array's (or `stackalloc`'s) span outlive its owner.** The storage is the struct (or the frame); the span is a window into it. Treat it like a `ref` to a local — the compiler catches the direct escape, but it cannot save you from a span you reasoned about as if it were a heap array. Copy out (to a `string` or pooled array) at the lifetime boundary.

Master these three and `stackalloc` plus `[InlineArray]` become what they are meant to be: a way to make the common case of small, ephemeral, fixed-size buffers cost nothing — no allocation, no GC, no indirection — while the worst case stays correct behind a single threshold check.

15 · SIMD and Vectorization: Vector, Vector128/256/512

Modern CPUs spend most of their silicon on width, not depth: a single core can apply one instruction to eight, sixteen, or sixty-four lanes of data at once. This chapter is about reaching that width from C# — first through the portable, JIT-managed `Vector<T>` and the fixed-width `Vector128/256/512<T>`, then through `System.Numerics.Tensors.TensorPrimitives` when you want the throughput without writing intrinsics by hand. The goal is not to memorize opcodes but to build correct, portable, *measurably faster* numeric kernels and to know precisely when SIMD pays and when it quietly loses.

The vectorization mental model

SIMD — Single Instruction, Multiple Data — is **data parallelism on one core**. It is orthogonal to threads. A thread pool spreads work across cores; SIMD widens the work each core does per instruction. They compose: a parallel `for` whose body is itself vectorized uses both axes. But the failure modes differ. Threading bugs are races and contention; SIMD bugs are forgotten tail elements, wrong horizontal reductions, and crashes on machines whose CPU lacks the instruction set you assumed.

The unit of work is a fixed-width register. On x86-64 these are XMM (128-bit), YMM (256-bit, AVX/AVX2), and ZMM (512-bit, AVX-512). On Arm64 they are the 128-bit NEON registers. A 256-bit register holds eight `float`s or four `double`s or thirty-two `byte`s. A vectorized loop processes one register-full per iteration, then a scalar **tail loop** mops up the `length % laneCount` remainder. That tail is not optional decoration — it is where most SIMD bugs live.

Three properties make a kernel a good SIMD candidate: the data is contiguous (a `Span<T>` or array, not a linked list), the operation is uniform across elements (no per-element branching that

diverges), and the arithmetic is associative enough that you can reorder it. Floating-point addition is *not* associative, so a vectorized sum produces a different — usually *better*, because partial sums stay smaller — result than a left-to-right scalar sum. You must decide that is acceptable before you vectorize; for most aggregations it is, but a bit-exact reproduction of a scalar reference is impossible.

Branching inside the loop is the property people most often get wrong. SIMD has no per-lane `if` ; instead you express conditionals as *masks*. A comparison like `Vector.GreaterThan(a, b)` returns a vector whose lanes are all-ones (`true`) or all-zeros (`false`), and `Vector.ConditionalSelect(mask, x, y)` blends two vectors lane-by-lane from that mask. Every lane still executes *both* arms of the conditional — you compute `x` and `y` for all lanes and then select — so branchy logic vectorizes only when both arms are cheap and side-effect-free. If one arm is an expensive call or a memory write, the mask approach computes it unconditionally for lanes that will discard it, and the vectorization loses. This is the deep reason divergent control flow defeats SIMD: there is no skipping lanes, only masking results.

`Vector<T>` : portable, variable width

`System.Numerics.Vector<T>` is the JIT's portable abstraction. Its width is **not known at compile time** — `Vector<T>.Count` is a JIT constant resolved per machine: `8` for `float` on an AVX2 box, `4` on plain Arm64 NEON. You write the loop once; the JIT picks the widest register it enables for the target (with one cap noted below). That portability is the whole point, and it is why `Vector<T>` should be your default for straight-line element-wise math.

One caveat catches people on AVX-512 hardware: `Vector<T>` does **not** automatically widen to 512 bits. By design the JIT caps it at 256 bits (so `Vector<float>.Count` stays `8`, not `16`) even when `Avx512F.IsSupported` , because AVX-512 can downclock the core (covered below). You opt into 512-bit `Vector<T>` explicitly with the `DOTNET_MaxVectorTBitWidth=512` environ-

ment variable (the same knob also caps the width downward, e.g. to `128`). If you genuinely want 512-bit width, prefer the fixed-width `Vector512<T>` and let `Vector512.IsHardwareAccelerated` make the call — that is what `TensorPrimitives` does internally.

The two workhorse APIs are `Vector.LoadUnsafe` (introduced in .NET 7) and `Vector.StoreUnsafe` , which move data between a `ref T` and a vector register without bounds checks. Each has a two-argument overload taking a `ref T` base and a `nuint` *element* offset, so you walk the buffer by advancing the offset rather than mutating the reference; if you prefer a moving cursor, `System.Runtime.CompilerServices.Unsafe.Add(ref r, i)` advances a `ref T` by `i` elements.

Here is a vectorized dot product over `ReadOnlySpan<float>` with an explicit scalar tail. This is the canonical kernel; internalize it and the rest of the chapter is variations.

```

using System;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

static float Dot(ReadOnlySpan<float> a,
    ReadOnlySpan<float> b)
{
    if (a.Length != b.Length)
        throw new ArgumentException("Spans must have
            equal length.");

    float total = 0f;
    int i = 0;

    if (Vector.IsHardwareAccelerated && a.Length >=
        Vector<float>.Count)
    {
        // Take refs to element 0; LoadUnsafe walks
            from there with no bounds checks.
        ref float ra = ref
            MemoryMarshal.GetReference(a);
        ref float rb = ref
            MemoryMarshal.GetReference(b);

        var acc = Vector<float>.Zero;
        int width = Vector<float>.Count;
        int lastBlock = a.Length - width; // index
            of the last full vector

        for (; i <= lastBlock; i += width)
        {
            var va = Vector.LoadUnsafe(ref ra,
                (nuint)i);
            var vb = Vector.LoadUnsafe(ref rb,
                (nuint)i);
            acc += va * vb; // fused
                element-wise multiply-accumulate
        }

        total = Vector.Sum(acc); //
            horizontal add of the lanes
    }

    // Scalar tail: handles length %
        Vector<float>.Count remaining elements.
    for (; i < a.Length; i++)
        total += a[i] * b[i];

    return total;
}

```

Several details earn their keep.

`Vector.IsHardwareAccelerated` is a JIT-folded constant; the whole SIMD block compiles away to nothing on a target with no SIMD support, leaving only the scalar tail — so this single method is correct and reasonable everywhere. This is the central JIT contract of the vector types: there is no IL opcode for “multiply two vectors.” The C# compiler emits an ordinary `call` to `Vector.op_Multiply`, and the *JIT* recognizes that method as an intrinsic and substitutes the native `vmulps` / `vmulpd` — a process called intrinsic recognition. A consequence is that `Vector<T>` math is only fast under a tiered or AOT JIT that performs this substitution; under the interpreter (used in some startup and debugging modes) the same code falls back to a genuine method call per lane and is *slower* than scalar. In Release with tiering, the recognition fires reliably. Another consequence: because `Vector<T>.Count` and `IsHardwareAccelerated` are JIT constants, the JIT can fully unroll small fixed-width loops and dead-eliminate the entire non-accelerated branch, leaving zero runtime overhead for the portability check. The `(nuint)i` offset is in *elements*, not bytes; `LoadUnsafe` scales by `sizeof(float)` internally. `Vector.Sum` (introduced in .NET 7) performs the horizontal reduction of the accumulator’s lanes into one scalar — before it existed you reduced manually, and we will see that pattern below for min/max where no `Vector.Sum` analogue applies directly.

The loop bound `i <= lastBlock` rather than `i < a.Length` is the crux of tail handling: it stops at the last index where a *full* vector still fits. Whatever is left — anywhere from `0` to `width - 1` elements — falls through to the scalar loop, which resumes from the same `i`. Get this off by one and you either read past the span (a hard-to-reproduce corruption with `LoadUnsafe`, which does *not* bounds-check) or silently drop the last few elements.

A note on **alignment**. `LoadUnsafe` / `StoreUnsafe` emit *unaligned* moves (`vmovups`), so the span’s data does not need to sit on a 32- or 64-byte boundary — they are correct at any address. On modern cores the penalty for unaligned access that

does not straddle a cache line is essentially zero, which is why portable code uses the unaligned form universally and does not bother with the manual-prologue dance of aligning to a boundary first. The one place alignment still bites is AVX-512 with the explicitly *aligned* load instructions (`vmovaps`), which fault on misaligned addresses — but you only reach those through the low-level intrinsics, never through `LoadUnsafe` . Stick to `LoadUnsafe` and alignment is a non-problem. The other latent cost is **GC compaction**: `MemoryMarshal.GetReference` returns a managed `ref` into a possibly-moveable array, and the `ref` is correctly tracked across collections, so no pinning is required — but if you instead obtain a raw pointer via `fixed` , you pin the array for the loop’s duration, adding fragmentation pressure on the GC heap. Prefer the `ref` -based `LoadUnsafe` over `fixed` +pointer kernels precisely to keep the array moveable.

Fixed-width vectors: `Vector128/256/512<T>`

`Vector<T>` hides the width; the fixed-width types expose it. `Vector256<float>` is *always* eight lanes wherever it compiles, regardless of whether the CPU has AVX2. That precision matters when an algorithm’s structure depends on the lane count — shuffles, gathers, lane-crossing permutes, or AVX-512 mask registers — which `Vector<T>` cannot express portably.

The cost of fixing the width is that *you* now own the

`IsHardwareAccelerated` check.

`Vector256.IsHardwareAccelerated` tells you whether 256-bit operations are genuinely accelerated (true on AVX2 hardware). If you emit `Vector256` math on a machine where it is not accelerated, the JIT may software-emulate it — slower than scalar — or, for the lower-level `System.Runtime.Intrinsics.X86` intrinsics, the instruction simply will not exist and you crash. **The fixed-width vector classes are safe to call unconditionally and will emulate; the ISA-specific intrinsic classes are not.** Always gate the latter on their `IsSupported` flag.

A `Vector256<float>` kernel with an explicit scalar fallback, squaring elements element-wise:

```

using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Intrinsics;

static void SquareInPlace(Span<float> data)
{
    int i = 0;

    if (Vector256.IsHardwareAccelerated && data.Length
        >= Vector256<float>.Count)
    {
        ref float r = ref
            MemoryMarshal.GetReference(data);
        int width = Vector256<float>.Count;
        // always 8 for float
        int lastBlock = data.Length - width;

        for (; i <= lastBlock; i += width)
        {
            Vector256<float> v =
                Vector256.LoadUnsafe(ref r, (nuint)i);
            Vector256.StoreUnsafe(v * v, ref r,
                (nuint)i);
        }

        // Scalar fallback AND tail in one loop: runs the
        // whole span when
        // Vector256 isn't accelerated, or just the
        // remainder when it is.
        for (; i < data.Length; i++)
            data[i] = data[i] * data[i];
    }
}

```

Note the single tail loop doubles as the *fallback path*. When `Vector256.IsHardwareAccelerated` is false, `i` stays `0` and the scalar loop processes everything. This is the idiomatic shape: one SIMD block, one scalar loop, no duplicated logic. The operator `*` on `Vector256<float>` lowers to a single `vmulps` on AVX2; `LoadUnsafe` / `StoreUnsafe` lower to `vmovups`. No allocations, no GC pressure — `Vector256<T>` is a `readonly struct` living in registers or on the stack.

When to drop to hardware intrinsics

`System.Runtime.Intrinsics.X86` (`Avx2`, `Avx512F`, `Sse42`, `Bmi1`, ...) and `System.Runtime.Intrinsics.Arm` (`AdvSimd`, `Crc32`, ...) expose individual CPU instructions one-to-one. You reach for them when the generic vector API has no operation for what you need: a specific shuffle, `Avx2.GatherVector256` for strided loads, `Avx512F` mask-based conditional stores, population counts, or carry-less multiply for CRC. Each class has an `IsSupported` static `bool` that is a JIT constant — branch on it and the dead path is eliminated.

The rule is absolute: **every ISA-specific intrinsic call must sit behind its `IsSupported` guard**. Reaching an unguarded `Avx2.Multiply` call on a CPU without AVX2 throws `PlatformNotSupportedException` — the JIT compiles the method body to a throw-helper stub, so the exception fires at runtime when that code is executed, not during compilation. The generic `Vector256.Multiply` does not throw — it emulates. So prefer the cross-platform `Vector128/256/512` API and descend to `Avx2` / `Avx512F` only for the specific instruction you cannot otherwise express, keeping the descent as small as possible:

```
using System.Runtime.Intrinsics;
using System.Runtime.Intrinsics.X86;

static Vector256<float>
    ReciprocalApprox(Vector256<float> v)
{
    // Fast ~12-bit reciprocal estimate; only Avx
    // exposes it directly.
    if (Avx.IsSupported)
        return Avx.Reciprocal(v);           // vrcpps –
        // one instruction, lower precision

    return Vector256<float>.One / v;        //
        // portable, full-precision division
}
```

`Avx.Reciprocal` (`vrcpps`) trades accuracy for speed and has no portable `Vector256` equivalent, which is exactly the kind of narrow, justified descent intrinsics are for. The

`Avx.IsSupported` guard means this method is safe to call on an Arm Mac — it falls back to true division.

AVX-512 and `Vector512<T>` : power, and the down-clocking trap

`Vector512<T>` (sixteen `float`s) maps to AVX-512. The headline subtlety, confirmed by the runtime team, is that **`Vector512.IsHardwareAccelerated` is conservative by design**. On Ice Lake and newer client/server parts, where AVX-512 does not meaningfully downclock the core, it reports `true`. On Skylake-X, Cascade Lake, and Cooper Lake — where heavy AVX-512 use drops the core frequency, sometimes hurting the *rest* of the program more than the kernel gains — it reports `false` *by default*, even though `Avx512F.IsSupported` still reports `true`. The platform is telling you: “the instructions exist, but using them at width 512 is likely a net loss here.”

The practical consequence: trust

`Vector512.IsHardwareAccelerated`, not `Avx512F.IsSupported`, when deciding whether to *prefer* 512-bit width. And measure. AVX-512 wins decisively on long, throughput-bound streams; on short workloads the downclock and the warm-up cost of the wide units can make `Vector256` — or even `Vector<T>`, which the JIT may *not* widen to 512 on downclock-prone hardware — the faster choice. A tiered selection:

```

using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Intrinsics;

static float SumTiered(ReadOnlySpan<float> data)
{
    int i = 0;
    float total = 0f;
    ref float r = ref
        MemoryMarshal.GetReference(data);

    if (Vector512.IsHardwareAccelerated && data.Length
        >= Vector512<float>.Count)
    {
        var acc = Vector512<float>.Zero;
        int w = Vector512<float>.Count;
        // 16
        for (; i <= data.Length - w; i += w)
            acc += Vector512.LoadUnsafe(ref r,
                (nuint)i);
        total += Vector512.Sum(acc);
    }
    else if (Vector256.IsHardwareAccelerated &&
        data.Length >= Vector256<float>.Count)
    {
        var acc = Vector256<float>.Zero;
        int w = Vector256<float>.Count;
        // 8
        for (; i <= data.Length - w; i += w)
            acc += Vector256.LoadUnsafe(ref r,
                (nuint)i);
        total += Vector256.Sum(acc);
    }

    for (; i < data.Length; i++)
        // shared tail
        total += data[i];

    return total;
}

```

In practice you rarely write this by hand —

`TensorPrimitives.Sum` already does exactly this tiering internally, and better. Which brings us to the API you should reach for first.

TensorPrimitives : the vectorized API you should usually use

`System.Numerics.Tensors.TensorPrimitives` (ship-quality since .NET 9, expanded in .NET 10, also available as a NuGet package for older targets) is a static class of vectorized span operations: `Sum`, `Dot`, `CosineSimilarity`, `Add`, `Multiply`, `Max`, `Min`, `Norm`, `Softmax`, `Sigmoid`, and dozens more.

Internally each one does the tiering, tail handling, and intrinsic selection you just saw — written and maintained by the people who wrote the JIT, validated on every supported ISA. For the common kernels, hand-rolling is strictly worse: more code, more bugs, and usually slower because the BCL implementations exploit instructions (FMA, masked loads) you would not bother with.

Most methods come in two forms: concrete `float` / `double` overloads, and a generic `T` overload constrained on the relevant numeric interface (`Sum<T> where T : IAdditionOperators<T,T,T>, IAdditiveIdentity<T,T>; CosineSimilarity<T> where T : IRootFunctions<T>`). The generic interfaces are part of .NET's generic-math story (C# 11 / .NET 7) and let one call site cover `float`, `double`, `Half`, and custom numeric types.

Cosine similarity — the bread and butter of embedding search — is a one-liner instead of three hand-tuned reductions:

```

using System;
using System.Numerics.Tensors;

// Two embedding vectors (e.g. from a text model).
float[] query = { 0.10f, 0.20f, 0.30f, 0.40f };
float[] doc   = { 0.12f, 0.18f, 0.33f, 0.39f };

float sim = TensorPrimitives.CosineSimilarity<float>
    (query, doc);
// Equivalent to Dot(q,d) / (sqrt(SumOfSquares(q)) *
//   sqrt(SumOfSquares(d))),
// but vectorized end-to-end with a single pass
// strategy.
Console.WriteLine($"cosine similarity = {sim:F4}");
// ~0.9971

// Plain reductions, same package:
float magnitude = TensorPrimitives.Norm<float>(query);
// L2 norm (Euclidean)
float total     = TensorPrimitives.Sum<float>(query);
// 1.00

```

`CosineSimilarity` throws if the spans differ in length or are empty (cosine similarity of a zero-length vector is undefined). It internally fuses the dot product and both magnitude accumulations, so it is faster than calling `Dot`, `Norm`, `Norm` separately, which would make three passes over memory. When you need an *element-wise* result rather than a reduction, the transform overloads take a destination span:

```

float[] a = { 1f, 2f, 3f, 4f, 5f, 6f, 7f, 8f };
float[] b = { 8f, 7f, 6f, 5f, 4f, 3f, 2f, 1f };
float[] dst = new float[a.Length];

TensorPrimitives.Add(a, b, dst);           // dst[i] =
    a[i] + b[i] → all 9
TensorPrimitives.Multiply(a, b, dst);      // dst[i] =
    a[i] * b[i]
TensorPrimitives.Sigmoid(a, dst);          // dst[i] =
    1 / (1 + e-a[i])

```

The destination may *exactly* alias a source (`TensorPrimitives.Add(a, b, a)` is legal and in-place), which the implementation handles correctly. What is not allowed is a *partial* overlap — a destination that overlaps an input but starts

at a different offset is undefined behavior, not a supported case. There is no allocation; you own `dst`.

Min/max reduction and manual horizontal folding

Reductions where the BCL gives you a vertical operator but not a `Sum`-style horizontal helper require you to fold the final accumulator's lanes yourself. `TensorPrimitives.Max / Min` exist and you should prefer them, but understanding the manual fold is worth it because lane-crossing reductions appear constantly in custom kernels. `Vector.Max / Vector.Min` (note: the static methods on `Vector`, taking two `Vector<T>` operands) combine two registers element-wise; after the loop you have a register of partial maxima and must collapse it to one scalar.


```

using System;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

static (float Min, float Max)
    MinMax(ReadOnlySpan<float> data)
{
    if (data.IsEmpty) throw new
        ArgumentException("Span is empty.");

    int i = 0;
    float min, max;

    if (Vector.IsHardwareAccelerated && data.Length >=
        Vector<float>.Count)
    {
        ref float r = ref
            MemoryMarshal.GetReference(data);
        int w = Vector<float>.Count;

        var vmin = Vector.LoadUnsafe(ref r, 0);
        var vmax = vmin;
        i = w;

        for (; i <= data.Length - w; i += w)
        {
            var v = Vector.LoadUnsafe(ref r,
                (nuint)i);
            vmin = Vector.Min(vmin, v);      //
            element-wise minimum, lane by lane
            vmax = Vector.Max(vmax, v);
        }

        // Horizontal fold: reduce the lanes of
        vmin/vmax to single scalars.
        min = vmin[0]; max = vmax[0];
        for (int lane = 1; lane < w; lane++)
        {
            if (vmin[lane] < min) min = vmin[lane];
            if (vmax[lane] > max) max = vmax[lane];
        }
    }
    else
    {
        min = max = data[0];
        i = 1;
    }

    for (; i < data.Length; i++)           // scalar
        tail / fallback continuation

```

```

    {
        float x = data[i];
        if (x < min) min = x;
        if (x > max) max = x;
    }

    return (min, max);
}

```

The horizontal fold over `w` lanes is a scalar loop, but it runs *once* after the vectorized loop, not per element, so its cost is amortized to nothing on long inputs. The indexer `vmin[lane]` on `Vector<T>` reads a single lane. For a faster lane-crossing reduction you would use log-step shuffles (`Vector256.Shuffle` then `Vector256.Min`, halving the active lanes each step), but the simple scalar fold is correct, readable, and negligible unless your inputs are tiny — in which case you should not be vectorizing at all. In production, `TensorPrimitives.Max(data)` and `TensorPrimitives.Min(data)` do all of this for you and handle NaN propagation per IEEE 754; reach for them first.

Integer kernels and the widening trap

Integer SIMD is in some ways simpler than float — addition *is* associative, so vectorized and scalar sums agree bit-for-bit — and in one way more dangerous: overflow is silent and lane-local. A `Vector<int>` accumulator of pixel values overflows at `int.MaxValue` per lane exactly as a scalar `int` would, but because the lanes wrap independently you can get a plausible-looking wrong answer rather than an obviously corrupt one. When the running total can exceed `int.MaxValue`, you must *widen* during accumulation.

The portable tool is `Vector.Widen`, which splits one `Vector<int>` into two `Vector<long>` halves (the low and high lanes), letting you accumulate into 64-bit lanes:

```

using System;
using System.Numerics;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

static long SumWidened(ReadOnlySpan<int> data)
{
    int i = 0;
    long total = 0;

    if (Vector.IsHardwareAccelerated && data.Length >=
        Vector<int>.Count)
    {
        ref int r = ref
            MemoryMarshal.GetReference(data);
        int w = Vector<int>.Count;
        var accLo = Vector<long>.Zero;           // two
            long-lane accumulators avoid
        var accHi = Vector<long>.Zero;           // 32-bit
            overflow during reduction

        for (; i <= data.Length - w; i += w)
        {
            var v = Vector.LoadUnsafe(ref r,
                (nuint)i);
            Vector.Widen(v, out Vector<long> lo, out
                Vector<long> hi);
            accLo += lo;
            accHi += hi;
        }
        total = Vector.Sum(accLo) + Vector.Sum(accHi);
    }

    for (; i < data.Length; i++)                // scalar
        tail accumulates into long
        total += data[i];

    return total;
}

```

`Vector.Widen` (available since the type's inception) is the safe default; for `byte` → `ushort` → `uint` chains the same pattern stacks. The scalar tail accumulates into the same `long total`, so the type discipline is consistent across both paths. The cost of widening is real — you now do twice the accumulator work — so apply it only when the value range demands it; a sum of small

bytes over a short span fits in `int` and needs no widening at all. Knowing your data's range is part of writing the kernel.

A worked example tying it together: normalized embedding scoring

A realistic kernel from a retrieval system: given a batch of document embeddings and one query, L2-normalize the query, then score every document by dot product against the normalized query, writing scores to an output span. This combines a transform, a reduction, and a per-row loop — and shows where `TensorPrimitives` and hand-written `Vector<T>` each fit.

```

using System;
using System.Numerics;
using System.Numerics.Tensors;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

static class EmbeddingScorer
{
    /// <summary>
    /// Scores each document row against the query
    /// using cosine similarity.
    /// docs is a row-major [rowCount × dim] flat
    /// buffer; scores has length rowCount.
    /// </summary>
    public static void Score(
        ReadOnlySpan<float> docs, ReadOnlySpan<float>
        query,
        Span<float> scores, int dim)
    {
        int rowCount = scores.Length;
        if (docs.Length != rowCount * dim)
            throw new ArgumentException("docs length
            must be rowCount * dim.");
        if (query.Length != dim)
            throw new ArgumentException("query length
            must equal dim.");

        for (int row = 0; row < rowCount; row++)
        {
            ReadOnlySpan<float> docRow =
            docs.Slice(row * dim, dim);
            // Let the BCL pick the widest accelerated
            // path and handle the tail.
            scores[row] =
            TensorPrimitives.CosineSimilarity<float>
            (query, docRow);
        }
    }

    /// <summary>
    /// Returns the index and score of the best-
    /// matching row.
    /// The reduction over scores is itself vectorized
    /// with Vector<float>.
    /// </summary>
    public static (int Index, float Score)
    ArgMax(ReadOnlySpan<float> scores)
    {
        if (scores.IsEmpty) throw new
        ArgumentException("No scores.");
    }
}

```

```

int best = 0;
float bestScore = scores[0];
int i = 1;

if (Vector.IsHardwareAccelerated &&
    scores.Length >= Vector<float>.Count * 2)
{
    ref float r = ref
MemoryMarshal.GetReference(scores);
    int w = Vector<float>.Count;
    var vmax = Vector.LoadUnsafe(ref r, 0);
    i = w;
    for (; i <= scores.Length - w; i += w)
        vmax = Vector.Max(vmax,
Vector.LoadUnsafe(ref r, (nuint)i));

    // Fold lanes, then a scalar argmax over
the (now small) candidate window.
    bestScore = vmax[0];
    for (int lane = 1; lane < w; lane++)
        if (vmax[lane] > bestScore) bestScore
= vmax[lane];

    // Find which index produced bestScore
(first occurrence).
    best = scores.IndexOf(bestScore);
    if (best < 0) best = 0; //
defensive: NaN edge cases
}

for (; i < scores.Length; i++)
{
    if (scores[i] > bestScore) { bestScore =
scores[i]; best = i; }
}

return (best, bestScore);
}
}

```

The division of labor is the lesson. The *similarity* computation is a textbook reduction the BCL already vectorizes optimally, so `CosineSimilarity` wins on both speed and correctness — writing it by hand would be reinventing tested code. The *argmax* over the resulting scores is a custom reduction (you need the index, not just the value) that `TensorPrimitives` does not directly expose with the index, so the hand-written `Vector<T>` fold is justified. Knowing which side of that line you are on is the entire

skill: lean on `TensorPrimitives` for standard kernels, descend to `Vector<T>` only when the operation is genuinely custom.

Benchmarking scalar vs. `Vector<T>`

VS. `TensorPrimitives`

Never assert a SIMD speedup without measuring it; the JIT, memory bandwidth, and input size all conspire to surprise you. BenchmarkDotNet is the standard tool.

```

using System;
using System.Numerics;
using System.Numerics.Tensors;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

[MemoryDiagnoser]
public class DotBenchmarks
{
    private float[] _a = null!, _b = null!;

    [Params(64, 1024, 65536)]
    public int N;

    [GlobalSetup]
    public void Setup()
    {
        var rng = new Random(42);
        _a = new float[N];
        _b = new float[N];
        for (int i = 0; i < N; i++) { _a[i] =
            (float)rng.NextDouble(); _b[i] =
            (float)rng.NextDouble(); }
    }

    [Benchmark(Baseline = true)]
    public float Scalar()
    {
        float s = 0f;
        for (int i = 0; i < _a.Length; i++) s += _a[i]
            * _b[i];
        return s;
    }

    [Benchmark]
    public float VectorT()
    {
        ReadOnlySpan<float> a = _a, b = _b;
        float total = 0f; int i = 0;
        if (Vector.IsHardwareAccelerated && a.Length
            >= Vector<float>.Count)
        {
            ref float ra = ref
                MemoryMarshal.GetReference(a);
            ref float rb = ref
                MemoryMarshal.GetReference(b);
            var acc = Vector<float>.Zero; int w =
                Vector<float>.Count;

```



```

        for (; i <= a.Length - w; i += w)
            acc += Vector.LoadUnsafe(ref ra,
                (nuint)i) * Vector.LoadUnsafe(ref rb,
                (nuint)i);
            total = Vector.Sum(acc);
    }
    for (; i < a.Length; i++) total += a[i] *
        b[i];
    return total;
}

[Benchmark]
public float Tensor() =>
    TensorPrimitives.Dot<float>(_a, _b);
}

// public static class Program { public static void
    Main() => BenchmarkRunner.Run<DotBenchmarks>
        (); }

```

Representative results on an AVX2 desktop (numbers vary by machine — *run it on yours*):

Method	N	Mean	Ratio	Allocated
Scalar	64	38.1 ns	1.00	—
VectorT	64	11.4 ns	0.30	—
Tensor	64	10.9 ns	0.29	—
Scalar	1024	612.7 ns	1.00	—
VectorT	1024	92.3 ns	0.15	—
Tensor	1024	78.5 ns	0.13	—
Scalar	65536	38_900.0 ns	1.00	—
VectorT	65536	9_950.0 ns	0.26	—
Tensor	65536	9_100.0 ns	0.23	—

Three readings worth absorbing. `TensorPrimitives` matches or beats the hand-written `Vector<T>` at every size — it uses FMA and may go to `Vector512` where the platform allows. At `N = 65536` all three converge toward memory bandwidth: the kernel is no longer compute-bound, so the SIMD advantage shrinks from the $\sim 7\times$ of `N = 1024` to $\sim 4\times$. And no implementation allocates — these are stack-and-register kernels, zero GC pressure, which is exactly what you want in a hot path. The `MemoryDiagnoser` confirming `Allocated = -` is part of the test, not a footnote.

Pitfalls

Forgetting the remainder. The most common SIMD bug. If `length` is not a multiple of `Vector<T>.Count`, the trailing `length % Count` elements must be processed by a scalar loop that resumes from the same index. Loop to `i <= length - width`, never `i < length`, in the vector loop, and always follow with `for (; i < length; i++)`. A unit test with a deliberately prime-length input (e.g. 1009 elements) will catch a dropped tail that a 1024-element test hides.

LoadUnsafe does not bounds-check. Unlike the span indexer, `Vector.LoadUnsafe(ref r, offset)` reads `Vector<T>.Count` elements starting at `offset` with no validation. An off-by-one in the bound reads adjacent memory — silent corruption, not an exception. Guard with the `length >= width` precondition and the correct `<= length - width` loop bound, and never let the offset exceed `length - width`.

Assuming Vector512 is always faster. AVX-512 can down-clock the core on Skylake-X/Cascade Lake/Cooper Lake, hurting the whole thread. That is why

`Vector512.IsHardwareAccelerated` returns `false` on those parts even though `Avx512F.IsSupported` is `true`. Gate width-512 *preference* on `Vector512.IsHardwareAccelerated`, and on short workloads benchmark against `Vector256` — the narrower width often wins because it avoids the frequency hit and the wide-unit warm-up. Wider is not automatically faster.

Intrinsics without IsSupported guards. Calling `Avx2.Multiply` or `Avx512F.Add` on hardware lacking that ISA throws `PlatformNotSupportedException`. Every `System.Runtime.Intrinsics.X86 / .Arm` call must sit behind `if (Avx2.IsSupported)` (a JIT constant, so the dead branch is eliminated). The cross-platform `Vector128/256/512` classes are exempt — they emulate when not accelerated — which is why they should be your default and the ISA-specific classes a deliberate, narrow exception.

Expecting bit-exact results. Vectorized reductions reassociate floating-point operations: a vector sum maintains `Count`

partial sums and folds them at the end, producing a different — often *more accurate* — result than a strict left-to-right scalar sum. Do not assert equality against a scalar reference; assert within a tolerance (`MathF.Abs(simd - scalar) < epsilon * n`). For integer kernels this is a non-issue (integer add is associative), but watch for overflow: a vectorized `int` sum overflows lane-by-lane exactly as scalar would, so widen to `long` accumulators when the total can exceed `int.MaxValue`.

When not to vectorize

SIMD is not free engineering. The kernel must be hot enough and long enough to repay the added code, the harder-to-read loop, and the test surface (tail, fallback, NaN, alignment). For spans of a few dozen elements the scalar loop is often within noise of the vectorized version and far easier to maintain — the benchmark at `N = 64` already shows the absolute saving is tens of nanoseconds. Branchy, divergent, or pointer-chasing logic does not vectorize: if each element needs a different decision, SIMD's uniform-instruction model fights you. And before writing a single intrinsic, check whether `TensorPrimitives` already has your operation — for sums, dots, norms, similarities, and element-wise math it almost certainly does, implemented better than you will manage and validated across every ISA the runtime supports. Hand-written `Vector<T>` earns its place only for genuinely custom reductions and transforms the BCL does not cover; reach for it last, measure always, and keep the scalar fallback correct for the day your code runs on a CPU you never tested.

Sources: [TensorPrimitives.CosineSimilarity \(Microsoft Learn\)](#), [TensorPrimitives.Sum \(Microsoft Learn\)](#), [Hardware Intrinsics in .NET 8 — AVX-512/Vector512 downclocking \(.NET Blog\)](#), [Vector.LoadUnsafe / StoreUnsafe \(Microsoft Learn\)](#).

16 · Struct Layout, Allocations, and the Garbage Collector

Performance in managed code is mostly a story about memory: how much you allocate, where it lands, how long it lives, and how the collector reclaims it. This chapter takes you from the byte-level layout of a single object — its header, its method table pointer, the padding the runtime inserts between fields — up through the generational heap, the SOH/LOH split, and the GC’s cost model, and finishes with the concrete techniques (escape analysis, pooling, pinning, allocation-free API design) that let you write hot paths that touch the collector as little as possible. The .NET 10 JIT is dramatically more aggressive about eliding allocations than any prior runtime, so part of the job is knowing what it now does for free.

The real memory cost of a small object

Every reference-type instance on the 64-bit CoreCLR heap carries two words of overhead before any of your fields. The first word is the **object header** (the sync block index — used for `lock`, `GetHashCode` identity, and a few CLR bookkeeping bits). The second is the **method table pointer** (often called the *type handle* or *MT*), which points at the type’s metadata: its *vtable*, its base type, its GC layout description, its interface map. Both are 8 bytes. The minimum object size on the 64-bit runtime is therefore not zero — it is 24 bytes: 8 (header) + 8 (MT) + 8 of payload slack, because the allocator rounds the total up so that even an “empty” object stays 8-byte aligned and leaves room for at least one pointer-sized field.

There is no *public* BCL API that returns the heap footprint of a reference-type instance. `RuntimeHelpers.GetObjectSize` exists in the runtime but is internal; `Unsafe.SizeOf<T>()` and the AOT-friendly `RuntimeHelpers.SizeOf(RuntimeTypeHandle)` (added in .NET 9) return the *element/local* size of `T` — for a reference type that is just the pointer width (8 bytes), not the object

it points at. So the per-instance figures below come from the layout rules above, not from a size call; to read an actual instance size at runtime, use a debugger (`!ObjSize` in SOS / `dotnet-dump`):

```
using System.Runtime.CompilerServices;

sealed class Empty { } // no
                        fields    -> 24 bytes on the heap (16
                        overhead + 8 slack)
sealed class OneInt { public int X; } // a
                        single 4-byte field -> still 24: the int fits
                        in the slack
sealed class TwoLongs { public long A, B; } // two 8-
                        byte fields    -> 16 overhead + 16 payload =
                        32 bytes

// Unsafe.SizeOf<T> for a reference type measures the
// *reference*, not the instance:
Console.WriteLine(Unsafe.SizeOf<Empty>()); // 8
// -- size of a reference, NOT the 24-byte
// instance
Console.WriteLine(Unsafe.SizeOf<OneInt>()); // 8
// For a value type it measures the inline payload,
// which is the number you usually want:
Console.WriteLine(Unsafe.SizeOf<int>()); // 4
```

The practical consequence: a `class Point { int X, Y; }` costs 24 bytes on the heap, of which only 8 are your data. Wrap a million of them in a `List<Point>` and you pay 24 MB of *objects* plus an 8 MB array of references pointing at them — 32 MB to store 8 MB of integers, scattered across the heap so every dereference is a likely cache miss. The `struct Point { int X, Y; }` version stores 8 contiguous bytes per element directly inside the backing array: 8 MB total, sequential, cache-friendly. This is the single most important reason to reach for structs in data-dense code, and it has nothing to do with avoiding the GC — it is about density and locality.

Struct field layout, padding, and alignment

A struct has no header and no method table pointer when it lives on the stack or inline in an array. Its size is the sum of its fields — but only after the runtime applies **alignment** rules, and those

can waste a surprising number of bytes if you order fields carelessly. Each primitive must sit at an offset that is a multiple of its own size (an `int` at a 4-byte boundary, a `long` or a reference at an 8-byte boundary), and the struct's total size is rounded up to a multiple of its largest field's alignment so that arrays of it stay aligned.

```
using System.Runtime.CompilerServices;

struct BadOrder    // byte, long, byte, int ->
    padding everywhere
{
    public byte A; // offset 0
    public long B; // must be 8-aligned -> 7 bytes
    padding after A, B at offset 8
    public byte C; // offset 16
    public int D;  // must be 4-aligned -> 3 bytes
    padding after C, D at offset 20
} // total rounded up to multiple
    of 8 -> 24 bytes

struct GoodOrder   // long, int, byte, byte ->
    descending size, holes filled
{
    public long B; // offset 0
    public int D;  // offset 8
    public byte A; // offset 12
    public byte C; // offset 13
} // 14 bytes of data, rounded up
    to 16

Console.WriteLine(Unsafe.SizeOf<BadOrder>()); // 24
Console.WriteLine(Unsafe.SizeOf<GoodOrder>()); // 16
```

`Unsafe.SizeOf<T>()` reports the *managed* size the way the type behaves in arrays and on the stack — this is the number you care about for density. Because a blittable struct keeps the field order you declare (see the layout subtlety below), the rule of thumb is **you order the fields yourself, largest to smallest** (references count as 8 bytes), so the small ones drop into the alignment holes the big ones would otherwise leave. Reordering `BadOrder` to `GoodOrder` cut the size by a third — across a 10-million-element array that is 80 MB you do not pay, and it is purely free.

A subtlety: the C# compiler emits

`[StructLayout(LayoutKind.Sequential)]` for a struct *by default*, so the field order you write is what you nominally get. But the runtime overrides that for any struct that **contains managed references** — such a struct is treated as `LayoutKind.Auto`, and the CLR is then free to reorder and pack its fields however it likes (which is why a struct with references is not blittable and cannot be passed to native code). Blittable structs (no references) keep their declared sequential order. So “order largest-first” matters most for all-blittable structs; for reference-containing structs the runtime already packs optimally, but you also lose the *guarantee* of a fixed layout. When you need a guaranteed layout — interop, memory-mapped data, wire formats — keep the struct blittable and rely on the default `Sequential` (or state it explicitly), optionally with `Pack` to control alignment.

Reinterpreting bytes with explicit layout

`[StructLayout(LayoutKind.Explicit)]` plus `[FieldOffset]` lets you overlap fields at chosen byte offsets — a C-style union. This is the canonical zero-cost way to reinterpret the bits of one type as another without copying. The classic use is splitting a wide primitive into its component bytes, or reading the raw bit pattern of a float.

```

using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
struct FloatBits
{
    [FieldOffset(0)] public float Value; // 4
    bytes, offsets 0..3
    [FieldOffset(0)] public uint Bits; // same 4
    bytes, viewed as unsigned
    [FieldOffset(0)] public byte B0; //
    individual bytes, little-endian
    [FieldOffset(1)] public byte B1;
    [FieldOffset(2)] public byte B2;
    [FieldOffset(3)] public byte B3;
}

var f = new FloatBits { Value = -2.0f };
Console.WriteLine($"0x{f.Bits:X8}");
// 0xC0000000
Console.WriteLine($"{f.B3:X2} {f.B2:X2} {f.B1:X2}
    {f.B0:X2}"); // C0 00 00 00
Console.WriteLine($"sign bit set: {(f.Bits >> 31) !=
    0}"); // True

```

Two hard rules with explicit layout. First, **a managed reference may never share its bytes with a non-reference field** — overlapping an `object` field with an `int`, or placing a reference at a misaligned offset, is rejected at type-load time with a `TypeLoadException`, because the GC scans the heap using each type’s reference map and must be able to tell a real pointer from an arbitrary bit pattern at every slot. Overlapping two *reference* fields at the same aligned offset is permitted (the GC simply sees one reference slot), but mixing a reference with anything else there is not; in practice, keep unions to unmanaged data. Second, explicit layout disables the runtime’s freedom to optimize, so use it deliberately. For the common “just reinterpret these bits” case where you do not need a named union, prefer `System.Runtime.CompilerServices.Unsafe.As<TFrom, TTo>(ref x)` or `BitConverter.SingleToUInt32Bits / MemoryMarshal.Cast<T,U>` — they express intent without a custom type and the JIT lowers them to nothing.

The generational heap: SOH, LOH, and the cost model

The CoreCLR GC is a **generational, mark-sweep-compact** collector. Newly allocated small objects land in **gen 0**, a small nursery. A gen 0 collection is cheap because most young objects are already dead (the *generational hypothesis*); survivors are promoted to gen 1, and survivors of gen 1 to **gen 2**, the long-lived generation. The collector compacts the **Small Object Heap (SOH)** — it slides survivors together to defragment, which is why managed allocation is just a pointer bump and why you rarely fragment the SOH.

Objects of **85,000 bytes or more** go to the **Large Object Heap (LOH)** instead. The LOH exists because copying large buffers during compaction is expensive, so by default the LOH is **swept but not compacted** — dead blocks are reclaimed in place and reused, but live blocks are not slid together. The LOH is collected only as part of a **gen 2** collection, which is the most expensive kind. Two cost facts follow directly:

- **Anything you put on the LOH is effectively gen 2.** It is not collected until a full gen 2 GC runs, so a short-lived 100 KB array keeps its memory committed far longer than a short-lived 100-byte one. Allocating large transient buffers in a loop is one of the most common managed-memory mistakes.
- **The LOH fragments.** Because it is not compacted by default, a workload that repeatedly rents and drops differently-sized large arrays leaves a Swiss-cheese heap: plenty of free bytes, but no single hole big enough for the next request, forcing the GC to grow the heap. You can request a one-time compaction with `GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce` before a known quiet point, but the right fix is usually to stop allocating large arrays at all — pool them (covered below).

```
// 85,000 bytes is the threshold. A byte[] crosses it
// at ~85,000 elements;
// for wider element types it crosses much sooner.
const int LohThreshold = 85_000;
Console.WriteLine(GC.GetGeneration(new
    byte[LohThreshold])); // 2 -> on the LOH
Console.WriteLine(GC.GetGeneration(new
    byte[LohThreshold - 100])); // 0 -> on the
    SOH
Console.WriteLine(GC.GetGeneration(new long[10_700]));
// 2 -> 10,700 * 8 ≈ 85,600 bytes
```

Server vs workstation, concurrent GC, and DATAS

There are two GC *flavors* and a concurrency setting layered on top:

- **Workstation GC** uses a single managed heap and tunes for low latency on a client process sharing the machine. It is the default for non-server console and desktop apps.
- **Server GC** creates one heap and one dedicated background collection thread **per logical core**, parallelizing both allocation and collection. It tunes for throughput, assuming the process owns the box. It is the default for ASP.NET Core. The tradeoff is higher baseline memory (N heaps) and more threads.
- **Concurrent (background) GC** runs gen 2 collection largely in parallel with your threads, trading a little throughput for much shorter pause times. It is on by default for both flavors and you almost never want it off.

Configure these in the project file or `runtimeconfig.json`, not in code:

```
<PropertyGroup>

    <ServerGarbageCollection>true</ServerGarbageCollection>
    <!-- DOTNET_gcServer=1 -->

    <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>
</PropertyGroup>
```

The major shift to know for .NET 10: **DATAS (Dynamic Adaptation To Application Sizes)** is on by default.

Introduced as opt-in in .NET 8 and made default for Server GC in .NET 9, DATAS in .NET 10 dynamically sizes the heap roughly proportional to the application's live data set rather than to the core count. The headline effect is that Server GC no longer reflexively grows to consume memory proportional to a 64-core box when the actual working set is small — a major win for containerized and bursty workloads where memory costs money and many processes share a node. If you previously pinned

`DOTNET_GCHeapCount` to tame Server GC memory on big hosts, re-measure on .NET 10 first; DATAS likely makes that knob unnecessary, and you can disable it with

`DOTNET_GCDynamicAdaptationMode=0` if you have a throughput-critical service that wants the old behavior. ([Microsoft Learn — DATAS](#))

For latency-sensitive request handling, the practical levers are: prefer Server GC for throughput, keep concurrent GC on, and for a brief critical section use

`GC.TryStartNoGCRegion(totalSize)` to pre-reserve a budget and suppress collection entirely — but only if you can bound your allocations, because exceeding the budget ends the region and the suppressed collection fires anyway.

Escape analysis and what the JIT elides

The cheapest allocation is the one that never happens. The .NET 10 JIT performs substantially deeper **escape analysis** than .NET 9: it proves that an object's lifetime never leaves the method (it does not escape into a field, a returned value, another

thread, or anything the GC must later see) and, when proven, allocates it **on the stack** instead of the heap. There is then no GC tracking, no header, no MT pointer write, and the storage vanishes when the frame returns.

What .NET 10 newly stack-allocates when it proves non-escape, on top of the struct-local and `Span<T>`-over-local cases that already worked: small fixed-size arrays of value types **and** — new in .NET 10 — small fixed-size arrays of *reference* types, plus the closure display-class objects and delegate objects behind non-escaping lambdas (also new in .NET 10, gated on the JIT being able to inline the consumer). The eligibility test is a total-size budget on the allocation plus a proof of non-escape, not a fixed per-type element count, and the JIT only stack-allocates arrays whose length is a compile-time constant — so treat “what gets stacked” as an optimization detail to verify, never a contract. (Microsoft Learn — What’s new in the .NET 10 runtime)

```
// In .NET 10 this method very likely allocates ZERO
// heap bytes:
// the int[] never escapes, so the JIT puts it on the
// stack.
static int SumSquares(int n)
{
    var buf = new int[4];           // candidate for
    // stack allocation
    for (int i = 0; i < 4; i++) buf[i] = i * i;
    int total = 0;
    foreach (var v in buf) total += v;
    return total;                  // only an int
    // escapes, not the array
}
```

You cannot *force* stack allocation — it is an optimization, not a guarantee — and three things defeat it: storing the object in a field or returning it, passing it somewhere the JIT cannot inline and analyze, and a variable size (`new int[n]` is harder than `new int[4]`). The reliable, explicit alternative for small local buffers is `stackalloc` into a `Span<T>` , which *guarantees* stack storage:

```
Span<byte> scratch = stackalloc byte[256]; // always
    on the stack, bounded size only
```

Never `stackalloc` a size you do not control — an attacker-influenced length overflows the stack. The idiomatic pattern is `stackalloc` for small bounded buffers and `ArrayPool` for anything larger or variable.

C# 14's implicit Span conversions and allocation-free APIs

Designing allocation-free APIs in C# 14 is markedly more pleasant because of **first-class Span support**, introduced in C# 14 (.NET 10). The compiler now knows the conversion relationships among `T[]`, `Span<T>`, and `ReadOnlySpan<T>` as *built-in implicit conversions*, so they participate in overload resolution, extension-method receiver binding, and generic type inference. A `string` converts implicitly to `ReadOnlySpan<char>`, an array to `Span<T>`, and `Span<T>` to `ReadOnlySpan<T>` covariantly — all without an explicit `.AsSpan()` call. ([Microsoft Learn — first-class span types](#))

```
// One span-based overload now serves arrays, strings,
    and stackalloc'd buffers.
static int CountVowels(ReadOnlySpan<char> text)
{
    int count = 0;
    foreach (char c in text)
        if (c is 'a' or 'e' or 'i' or 'o' or 'u')
            count++;
    return count;
}

CountVowels("hello world"); // string
    -> ReadOnlySpan<char>, no alloc (C# 14)
CountVowels(new[] { 'a', 'b', 'c' }); // char[]
    -> ReadOnlySpan<char>, no alloc
Span<char> tmp = stackalloc char[3];
tmp[0] = 'i'; tmp[1] = 'o'; tmp[2] = 'x';
CountVowels(tmp); //
    Span<char> -> ReadOnlySpan<char>
```

Before C# 14 you would have written separate `string`, `char[]`, and span overloads, or forced callers to write `.AsSpan()`. Accepting `ReadOnlySpan<T>` is now the default shape for a read-only, allocation-free input parameter, and the language stops fighting you about it.

Proving allocations are gone: before/after to zero

Intuition about allocations is unreliable; measure.

`GC.GetAllocatedBytesForCurrentThread()` returns a monotonically increasing per-thread counter of bytes allocated on the managed heap — diff it across an operation to get an exact, deterministic allocation count with no profiler attached. Here is a realistic refactor: formatting a key-value pair into a buffer, first with `string` concatenation (allocations), then with span formatting (zero).

```

using System.Globalization;

// BEFORE: string interpolation allocates the result
//          string and intermediate boxes.
static string FormatBefore(int id, double value)
    => $"id={id};value={value.ToString("F2",
    CultureInfo.InvariantCulture)}";

// AFTER: format into a caller-provided span, return
//          the written length. Zero heap.
static bool TryFormatAfter(int id, double value,
    Span<char> dest, out int written)
{
    written = 0;
    if (!"id=".AsSpan().TryCopyTo(dest)) return false;
    int pos = 3;
    if (!id.TryFormat(dest[pos..], out int n,
        provider: CultureInfo.InvariantCulture))
        return false;
    pos += n;
    if (pos + 7 > dest.Length) return false;
    ";value=".AsSpan().CopyTo(dest[pos..]); pos += 7;
    if (!value.TryFormat(dest[pos..], out n, "F2",
        CultureInfo.InvariantCulture)) return false;
    written = pos + n;
    return true;
}

long b0 = GC.GetAllocatedBytesForCurrentThread();
_ = FormatBefore(42, 3.14159);
long b1 = GC.GetAllocatedBytesForCurrentThread();

Span<char> buffer = stackalloc char[64];
b1 = GC.GetAllocatedBytesForCurrentThread();
_ = TryFormatAfter(42, 3.14159, buffer, out _);
long b2 = GC.GetAllocatedBytesForCurrentThread();

Console.WriteLine($"before: {b1 - b0} bytes"); //
// before: ~96 bytes (result string + the "3.14"
// from ToString)
Console.WriteLine($"after: {b2 - b1} bytes"); //
// after: 0 bytes

```

The `before` path allocates the result `string` (plus, depending on the formatter, a temporary). The `after` path writes characters into a stack buffer and hands back a length — nothing reaches the heap. Note that `TryFormat` on primitives (`int`, `double`,

`DateTime` , `Guid`) is the workhorse of allocation-free formatting and exists on every numeric and date type in the BCL.

For systematic benchmarking, **BenchmarkDotNet's** `[MemoryDiagnoser]` reports allocated bytes and GC collections per generation alongside timing. It is the standard tool; treat the `Allocated` column as a regression gate.

```
using BenchmarkDotNet.Attributes;

[MemoryDiagnoser] // adds
                  Gen0/Gen1/Gen2 and Allocated columns
public class FormatBench
{
    [Benchmark(Baseline = true)]
    public string Before() => FormatBefore(42,
        3.14159);

    [Benchmark]
    public int After()
    {
        Span<char> buf = stackalloc char[64];
        TryFormatAfter(42, 3.14159, buf, out int n);
        return n;
    }
    // Allocated column: Before ~96 B, After 0 B.
}
```

For production diagnosis where you cannot rebuild with attributes, `dotnet-trace` with the GC providers captures allocation tick events and GC pauses from a live process (`dotnet-trace collect --providers Microsoft-DotNETCore-SampleProfiler,Microsoft-Windows-DotNETRuntime:0x1:5 -p <pid>`), and `dotnet-counters monitor System.Runtime` gives you live gen-collection rates and heap sizes with no instrumentation.

ArrayPool on a serialization hot path

When you genuinely need a large or variably-sized buffer on a hot path, do not allocate it — **rent it from** `ArrayPool<T>` and return it. The shared pool (`ArrayPool<T>.Shared`) is a thread-aware, bucketed cache of arrays sized to powers of two; renting

reuses an existing array (no allocation, no LOH pressure) and returning makes it available again. The non-negotiable discipline is a `try / finally` so the buffer comes back even on exception, and **never reading or writing the array after you return it**.

```
using System Buffers;
using System Text;

static byte[] SerializeFrame(ReadOnlySpan<char>
    payload)
{
    // Rent a buffer big enough for the worst case
    // (UTF-8 is up to 3 bytes/char for BMP).
    int maxBytes =
        Encoding.UTF8.GetMaxByteCount(payload.Length)
        + 4; // +4 length prefix
    byte[] rented =
        ArrayPool<byte>.Shared.Rent(maxBytes);
    try
    {
        int bodyLen = Encoding.UTF8.GetBytes(payload,
            rented.AsSpan(4));
        // little-endian length prefix

        System.Buffers.Binary.BinaryPrimitives.WriteInt32LittleEndian(rented, bodyLen);

        // Copy exactly the bytes we used into a
        // right-sized result.
        // (If the caller could also take a
        // span/IBufferWriter, we'd skip even this.)
        return rented.AsSpan(0, bodyLen +
            4).ToArray();
    }
    finally
    {
        // clearArray: true zeroes the buffer first -
        // do this if it held secrets,
        // otherwise leave it false to avoid the
        // memset cost.
        ArrayPool<byte>.Shared.Return(rented,
            clearArray: false);
    }
}
```

`Rent(n)` returns an array of **at least** `n` elements — frequently larger — so always track the real length yourself and slice; never

assume `rented.Length == n`. Three pitfalls bite people repeatedly:

- **Forgetting to return** turns the pool into a slow leak: the pool re-allocates to satisfy the next rent, and you get the worst of both worlds (allocation *and* a growing pool). The `try` / `finally` is mandatory, not stylistic.
- **Returning a buffer still referenced elsewhere** is a use-after-free in managed clothing. If you returned `rented` directly to a caller instead of copying, and then `Return` ed it, a later rent hands the same array to unrelated code that scribbles over your “result.” Either copy out (as above) or do not return until the caller is done — never both.
- **Returning twice** corrupts the pool’s free list; the same array can then be handed to two callers simultaneously.

When the consumer can accept a span or implement `IBufferWriter<T>`, design the API to write into *its* buffer and skip the final `.ToArray()` entirely — that is the true zero-allocation shape.

ObjectPool across a request loop

`ArrayPool` handles arrays; for reusable *objects* with expensive construction or internal buffers — `StringBuilder`, `MemoryStream`, parser state machines — use `Microsoft.Extensions.ObjectPool`. A `StringBuilder` that grows to a few KB is exactly the kind of object you want to reset and reuse rather than re-grow per request.

```

using Microsoft.Extensions.ObjectPool;
using System.Text;

// Built-in policy clears the builder on return and
// discards over-large ones
// so a single huge request doesn't permanently bloat
// the pool.
var provider = new DefaultObjectPoolProvider {
    MaximumRetained = 64 };
ObjectPool<StringBuilder> pool =
    provider.CreateStringBuilderPool(initialCapacity:
        256, maximumRetainedCapacity: 16 * 1024);

string BuildResponse(IReadOnlyList<(string Key, int
    Value)> items)
{
    StringBuilder sb = pool.Get();           // reused
                                              instance, already-warm capacity
    try
    {
        foreach (var (key, value) in items)

            sb.Append(key).Append('=').Append(value).Append(';');
        return sb.ToString();                // the one
                                              unavoidable allocation: the result
    }
    finally
    {
        pool.Return(sb);                     // policy
                                              resets length, keeps capacity
    }
}

```

Benchmarked against `new StringBuilder()` per call in a request loop, the pooled version eliminates the builder object **and** its internal char-array chunks from the per-request allocation profile — under `[MemoryDiagnoser]` you see the `Allocated` column drop to just the final `ToString()` result, and `Geno` collection frequency fall correspondingly. The tradeoffs: a pool adds a tiny `Get / Return` synchronization cost, it retains memory (those 64 warm builders stay alive), and a pooled object handed out and never returned silently degrades the pool to fresh allocation. Pool only objects whose construction cost or buffer reuse actually justifies it; pooling a trivial struct-like object is pure overhead.

Pinning, GCHandle, and fixed for interop

The compacting GC moves objects, so any time you hand a managed buffer's address to native code, the buffer must be **pinned** — held in place so the collector will not relocate it mid-call.

Three mechanisms, in increasing scope:

- `fixed` pins a managed array or string for the duration of a statement block. Cheapest and safest; the pin is scoped and cannot leak. Use it for synchronous P/Invoke where the native call returns before the block exits.
- `GCHandle.Alloc(obj, GCHandleType.Pinned)` pins for an *arbitrary* duration — necessary when native code holds the pointer across calls (a callback registration, an async native operation). You **must** call `handle.Free()`, ideally in a `finally`, or you pin that object forever.
- `Memory<T>.Pin()` returns a `MemoryHandle` (disposable) and is the modern, `Memory`-friendly equivalent.

```

using System.Runtime.InteropServices;

// Synchronous interop: fixed scopes the pin to
// exactly the native call.
static unsafe void HashInPlace(byte[] data)
{
    fixed (byte* p = data) // pinned
                           // only inside this block
    {
        NativeHash(p, data.Length); // returns
                                     // before the block exits -> safe
    }
}

// Long-lived pin: native side keeps the pointer past
// this method.
static GCHandle RegisterBuffer(byte[] buffer)
{
    var handle = GCHandle.Alloc(buffer,
        GCHandleType.Pinned);
    NativeRegister(handle.AddrOfPinnedObject(),
        buffer.Length);
    return handle; // caller
                  // MUST Free() when native is done
}

[DllImport("native")] static extern unsafe void
    NativeHash(byte* data, int len);
[DllImport("native")] static extern void
    NativeRegister(nint addr, int len);

```

The fragmentation hazard: **a pinned object cannot be moved, so it blocks compaction of the region it sits in.** A long-lived pin in the middle of gen 0/1 forces the GC to leave a hole and route survivors around it, fragmenting the SOH — the one heap that is normally pristine because it compacts. Pin for as short a time as possible (`fixed` over `GCHandle` whenever the lifetime allows), and for buffers that *must* stay pinned long-term, allocate them on the **Pinned Object Heap (POH)** via `GC.AllocateArray<T>(length, pinned: true)` — the POH is a dedicated region for pinned data so your pins do not perforate the moving heap.

```
// .NET 5+: a buffer that is permanently pinned,
// isolated on the POH.
byte[] ioBuffer = GC.AllocateArray<byte>(4096, pinned:
    true);
// Safe to hand &ioBuffer[0] to native code for the
// lifetime of the array
// without fragmenting gen 0/1 – it never lived there.
```

The hidden allocations: boxing, closures, and defensive copies

The most dangerous allocations are the ones you do not see, because the type system says “struct” while the JIT quietly heap-allocates. Three sources dominate.

Boxing. Assigning a value type to `object`, to a non-generic interface reference, or passing it where `object` is expected, allocates a boxed copy on the heap. The trap is interfaces: calling an interface method on a struct through an interface-typed variable boxes it.

```
int x = 5;
object boxed = x;           // ALLOCATES a
                             boxed int (24 bytes)
IComparable c = x;         // ALSO boxes
// foreach over a struct enumerator exposed only via
// IEnumerator<T> boxes the enumerator.
```

Closure capture. A lambda or local function that captures a local variable lifts that variable into a compiler-generated *display class* — a heap object — and the delegate itself is an allocation.

In .NET 10 the JIT *can* stack-allocate the closure if it proves the delegate never escapes the method, but the moment the delegate is stored, returned, or passed to something opaque, the closure goes to the heap.

```
int threshold = 10;
// This captures `threshold` into a display class; if
// `Where` keeps the delegate
// alive (it does, lazily), the closure escapes and is
// heap-allocated.
var filtered = numbers.Where(n => n > threshold);
```

Defensive copies of readonly structs. When you call a non-`readonly` instance method on a `readonly` field or an `in` parameter of struct type, the compiler must assume the method might mutate `this`, so it makes a **defensive copy** before the call — silently, on every call, sometimes in a loop. The fix is to mark the struct `readonly struct` (so the compiler knows no method mutates) or mark individual members `readonly`. This is invisible in source and shows up only as mysterious copy overhead in the disassembly.

```
readonly struct Vector3      // readonly struct => no
                             defensive copies through `in`/readonly
{
    public readonly float X, Y, Z;
    public Vector3(float x, float y, float z) => (X,
        Y, Z) = (x, y, z);
    public float Length() => MathF.Sqrt(X * X + Y * Y
        + Z * Z); // safe to call via `in`
}

static float Sum(in Vector3 v) => v.Length(); // no
                                             copy: Vector3 is a readonly struct
```

C# 14 adds a complementary tool: **user-defined compound assignment operators** (introduced in C# 14, .NET 10), declared as `void`-returning *instance* operators (`public void operator +=(T other)`) that mutate `this` in place instead of returning a new value. For a large mutable struct or a class wrapping a buffer, this avoids constructing a fresh instance on every `+=`, cutting both allocation and copy cost on accumulation hot paths. ([Microsoft Learn — user-defined compound assignment](#))

```

struct BigAccumulator
{
    public long Sum;
    public int Count;
    // Instance compound operator (C# 14): mutates in place, allocates nothing.
    public void operator +=(int value) { Sum += value; Count++; }
}

var acc = new BigAccumulator();
foreach (int v in data) acc += v; // in-place; no temporary instances created

```

Worked example: a zero-allocation log line aggregator

This pulls the chapter together — pooled buffers, span formatting, struct density, and

`GC.GetAllocatedBytesForCurrentThread()` to prove the result. The task: aggregate a batch of integer metrics into a single semicolon-delimited line, on a hot path called per request, with zero per-call heap allocation beyond the final string the caller demanded.


```

using System.Buffers;
using System.Globalization;

readonly record struct Metric(int Code, double Value);
    // 16 bytes, lives inline in the array

static class MetricFormatter
{
    // Caller owns the result string; everything else
    // is rented or stack.
    public static string Format(ReadOnlySpan<Metric>
        metrics)
    {
        // Worst case: each metric ~ "9999=-1234.56;"
        // ~ 24 chars. Rent, don't allocate.
        char[] rented =
            ArrayPool<char>.Shared.Rent(metrics.Length *
                24 + 1);
        try
        {
            Span<char> dest = rented;
            int pos = 0;
            foreach (ref readonly Metric m in metrics)
                // ref readonly: no per-element copy
                {
                    if (!m.Code.TryFormat(dest[pos..], out
                        int n, provider:
                        CultureInfo.InvariantCulture))
                        throw new
                        InvalidOperationException("buffer too small");
                    pos += n;
                    dest[pos++] = '=';
                    if (!m.Value.TryFormat(dest[pos..],
                        out n, "F2", CultureInfo.InvariantCulture))
                        throw new
                        InvalidOperationException("buffer too small");
                    pos += n;
                    dest[pos++] = ';';
                }
            return new string(dest[..pos]);
            // the single, unavoidable allocation
        }
        finally
        {
            ArrayPool<char>.Shared.Return(rented);
        }
    }
}

// --- proof ---
Metric[] batch = [ new(101, 3.14), new(102, 2.71),
    new(103, -9.5) ];

```

```

// warm up the JIT and the pool so the measurement
// reflects steady state
_ = MetricFormatter.Format(batch);

long before = GC.GetAllocatedBytesForCurrentThread();
string line = MetricFormatter.Format(batch);
long after = GC.GetAllocatedBytesForCurrentThread();

Console.WriteLine(line); //
    101=3.14;102=2.71;103=-9.50;
Console.WriteLine($"allocated: {after - before}
    bytes");
// allocated: 80 bytes -> ONLY the result string; the
// working buffer was pooled.

```

Every design choice here is deliberate. `Metric` is a `readonly record struct`, so the input array stores 16-byte elements in-line — no per-element heap object, no indirection, cache-friendly iteration. `foreach (ref readonly Metric m ...)` iterates by reference into the array, so there is no defensive copy of each struct (it is `readonly`, so the JIT also knows the body cannot mutate it). The working buffer comes from `ArrayPool<char>.Shared` inside a `try / finally`, so it is reused across calls and never touches the LOH even when batches are large. `int.TryFormat` and `double.TryFormat` write digits straight into the span with zero intermediate strings. The only heap allocation per call is the result `string` the caller asked for — and if the caller could accept a `ReadOnlySpan<char>` or write into its own `IBufferWriter<char>`, even that would disappear. The 80-byte figure is exactly the heap cost of that 28-character string: 16 bytes of object overhead (header + method-table pointer) + 4-byte length + 28×2 bytes of UTF-16 chars + a 2-byte null terminator = 78, rounded up to the 8-byte alignment boundary. Nothing else leaked.

Putting it together

The mental model to carry out of this chapter: managed memory performance is dominated by *allocation rate* and *object lifetime*, not raw collection speed. Keep small data in structs ordered largest-field-first for density; keep transient large buffers off the

LOH by pooling them; let the .NET 10 JIT's escape analysis erase the allocations it can, and use `stackalloc` , `Span<T>` , and C# 14's implicit span conversions to design APIs that never allocate in the first place. Trust nothing —

`GC.GetAllocatedBytesForCurrentThread()` , `[MemoryDiagnoser]` , and `dotnet-trace` turn allocation from a guessing game into a number you can regression-gate. And remember the silent allocators — boxing, closure capture, and defensive copies of non-`readonly` structs — because those are the ones that survive code review and show up only when you finally read the disassembly.

Sources consulted for C# 14 / .NET 10 specifics: [Microsoft Learn — First-class span types](#), [Microsoft Learn — What's new in the .NET 10 runtime](#), [Microsoft Learn — DATAS](#), [Microsoft Learn — User-defined compound assignment operators](#).

Part V — Concurrency and Asynchrony

17 · `async/await` Internals and the State Machine

`async` / `await` is syntactic sugar over a cooperative continuation-passing transform the compiler performs at the method level. Understanding the lowered shape — a struct state machine driven by `MoveNext`, glued to a *builder* and a chain of *awaiters* — is the difference between writing `async` code that merely runs and writing `async` code whose throughput, allocation profile, and deadlock behavior you can reason about deterministically. This chapter dissects every moving part, builds a custom awaitable from scratch, reads the compiler-generated IL shape, and reproduces the two failure modes that still ship in production daily: the `.Result` deadlock and the exception-swallowing `async void`.

The shape of the transform

When you mark a method `async`, the compiler does *not* emit a method whose body runs to completion. It emits a tiny stub that allocates (or stack-initializes) a generated **state machine** type, hands it to a **method builder**, and calls `builder.Start(ref stateMachine)`. The original method body is moved verbatim into the state machine's `MoveNext` method, with every `await` split into a *resume point* keyed by an integer `state` field.

Three contracts make this work, and you should know all three by name:

- **`IAsyncStateMachine`** — the generated type implements this. It has `void MoveNext()` (the resumable body) and `void SetStateMachine(IAsyncStateMachine)` (used when the struct is boxed to the heap on first suspension).
- **The builder** — `AsyncTaskMethodBuilder`, `AsyncTaskMethodBuilder<T>`, `AsyncValueTaskMethodBuilder<T>`, or `AsyncVoidMethodBuilder`, selected by your return type.

The builder owns the returned `Task / ValueTask` and bridges awaiter completion back to `MoveNext`.

- **The awaiter** — whatever `GetAwaiter()` returns on the thing you `await`. It must expose `bool IsCompleted`, `void OnCompleted(Action)` (and ideally `void UnsafeOnCompleted(Action)` via `ICriticalNotifyCompletion`), and `T GetResult()` (or `void GetResult()`).

The control flow on each `await expr;` is:

```
var awaiter = expr.GetAwaiter();
if (!awaiter.IsCompleted)
{
    state = /* resume label */;
    builder.AwaitUnsafeOnCompleted(ref awaiter, ref
        this); // suspend
    return;
    // MoveNext exits
}
// either fell through synchronously, or re-entered
// here on resume:
var result = awaiter.GetResult(); // observes value
// AND rethrows captured exception
```

That `if (!IsCompleted)` branch is the single most important line in the whole model. **`await` does not always yield.** If the awaiter reports completion synchronously, `MoveNext` falls straight through to `GetResult()` and keeps running on the current thread with zero scheduling. This is a feature (avoid pointless context switches) and a trap (a method full of `await`s on already-completed tasks can monopolize a thread and starve other work). More on that in *Pitfalls*.

Awaitables and awaiters: the duck-typed protocol

There is no `IAwaitable` interface. `await x` compiles if `x.GetAwaiter()` (instance method or extension member — see C# 14 extension blocks below) returns a type satisfying the *awaiter pattern*. That is deliberately duck-typed so the BCL,

third parties, and you can plug in custom suspension behavior. The full awaiter surface:

```
public interface INotifyCompletion           //
    System.Runtime.CompilerServices
{
    void OnCompleted(Action continuation);
}
public interface ICriticalNotifyCompletion :
    INotifyCompletion
{
    void UnsafeOnCompleted(Action continuation); //
    does NOT flow ExecutionContext
}
```

An awaiter must provide `bool IsCompleted { get; }`, `T GetResult()` (or `void GetResult()`), and implement `INotifyCompletion`. Implement `ICriticalNotifyCompletion` too — the builder prefers `AwaitUnsafeOnCompleted`, which skips a redundant `ExecutionContext` capture because the builder already flows it once for the whole method. Skipping that double-capture is a measurable allocation win on hot paths.

A complete custom awaitable: `YieldToThreadPool`

Here is a real, compilable awaitable. `await YieldToThreadPool.Instance` forces the continuation onto a thread-pool thread, unconditionally — unlike `Task.Yield()`, which posts back to the captured `SynchronizationContext` if one exists. This is the kind of primitive you build when you want to *escape* a UI or ASP.NET-legacy context on purpose.

```

using System.Runtime.CompilerServices;

public readonly struct YieldToThreadPool
{
    public static YieldToThreadPool Instance =>
        default;

    // The awaitable's only job: hand back an awaiter.
    public Awaiter GetAwaiter() => default;

    public readonly struct Awaiter :
        ICriticalNotifyCompletion
    {
        // false => the compiler always takes the
        suspend path, so the
        // continuation is guaranteed to run on a pool
        thread, never inline.
        public bool IsCompleted => false;

        public void GetResult() { } // nothing to
        observe, no exception to rethrow

        // Flows ExecutionContext (AsyncLocal, etc.) -
        the safe default.
        public void OnCompleted(Action continuation)
            =>
            ThreadPool.QueueUserWorkItem(static s =>
                s(), continuation, preferLocal: false);

        // The builder calls this one. We deliberately
        do NOT re-flow
        // ExecutionContext here; the async method
        builder already captured it.
        public void UnsafeOnCompleted(Action
            continuation) =>
            ThreadPool.UnsafeQueueUserWorkItem(static
                s => s(), continuation, preferLocal: false);
    }
}

```

Usage and why it matters:


```

async Task ProcessAsync()
{
    // ... runs on whatever thread / context we
    // started on ...
    await YieldToThreadPool.Instance;
    // Guaranteed here: a ThreadPool thread, no
    // SynchronizationContext.
    Debug.Assert(SynchronizationContext.Current is
        null);
    CpuBoundWork();
}

```

Note the strongly-typed `Awaiter` struct return — `ThreadPool.QueueUserWorkItem<TState>` (the generic, non-boxing overload that takes state plus `preferLocal`) avoids a closure allocation by passing `continuation` as state to a `static` lambda. The `readonly struct` `awaiter` means the whole thing is allocation-free up to the point the builder boxes the state machine on first real suspension.

The lowered state machine, by hand

Take a minimal async method:

```

async Task<int> SumAsync()
{
    int a = await GetAAsync();
    int b = await GetBAsync();
    return a + b;
}

```

Paste this into [SharpLab](#) with “Results: C#” to see the lowering. What the compiler emits is structurally this (cleaned up, names normalized, but faithful to the real shape on .NET 10):

```
[AsyncStateMachine(typeof(SumAsyncStateMachine))]
[DebuggerStepThrough]
Task<int> SumAsync()
{
    var sm = new SumAsyncStateMachine
    {
        builder =
            AsyncTaskMethodBuilder<int>.Create(),
        state = -1, // -1 ==
            "not started / running"
        @this = this
    };
    sm.builder.Start(ref sm); //
        synchronously runs MoveNext to first suspend
    return sm.builder.Task; // the hot
        task handed back to the caller
}

[CompilerGenerated]
private struct SumAsyncStateMachine :
    IAsyncStateMachine
{
    public int state;
    public AsyncTaskMethodBuilder<int> builder;
    public C @this; // captured
        'this'
    private int a; // hoisted
        local: lives across an await
    private TaskAwaiter<int> u1; // the
        awaiter we're parked on

    public void MoveNext()
    {
        int result;
        try
        {
            TaskAwaiter<int> awaiter;
            switch (state)
            {
                default: // state == -1, first entry
                    awaiter =
                        @this.GetAsync().GetAwaiter();
                    if (!awaiter.IsCompleted)
                    {
                        state = 0;
                        u1 = awaiter;

                        builder.AwaitUnsafeOnCompleted(ref awaiter,
                            ref this);

                        return; // SUSPEND
                    }
                #1
            }
        }
    }
}
```

```

    }
    goto AfterA;
    case 0: // resumed
after GetAAsync
    awaiter = u1; // reload
the hoisted awaiter
    u1 = default;
    state = -1;
    goto AfterA;
    case 1: // resumed
after GetBAsync
    awaiter = u1; // MUST
reload here too - the local
    u1 = default; //
'awaiter' is fresh on each MoveNext
    state = -1;
    goto AfterB;
}

AfterA:
    a = awaiter.GetResult(); // observe
+ rethrow if faulted
    {
        TaskAwaiter<int> awaiter2 =
@this.GetBAsync().GetAwaiter();
        if (!awaiter2.IsCompleted)
        {
            state = 1;
            u1 = awaiter2;
            builder.AwaitUnsafeOnCompleted(ref
awaiter2, ref this);
            return; // SUSPEND
#2
        }
        awaiter = awaiter2;
    }

AfterB:
    int b = awaiter.GetResult();
    result = a + b;
}
catch (Exception ex)
{
    state = -2; // -2 ==
done
    builder.SetException(ex); // faults
the returned Task
    return;
}
state = -2;
builder.SetResult(result); //
completes the returned Task with the value
}

```

```

void
    IAsyncStateMachine.SetStateMachine(IAsyncState
Machine boxed) =>
    builder.SetStateMachine(boxed);
}

```

Things worth internalizing from this lowering:

- **The state machine is a `struct`**. On the synchronous fast path (every awaited task already complete) it never leaves the stack — zero heap allocation. Only on the *first* real suspension does the builder box it onto the heap (then call `SetStateMachine` to repoint the boxed copy's builder at itself) so the continuation has something durable to resume into. This is why a method that usually completes synchronously is cheap, and one that always suspends pays one box.
- **Locals that live across an `await` are hoisted to fields** (`a` here). Locals that don't (`b`) stay as `MoveNext` locals. This is also why a `ref` local that crosses an `await` is a problem — a field can't be a `ref` to managed stack memory. (See the C# 13 section.)
- **`builder.AwaitUnsafeOnCompleted` wires the awaiter's continuation back to `MoveNext`**. On the *first* suspension it boxes the state machine and captures the current `ExecutionContext` into that box; on later suspensions it reuses the same box and capture. The awaiter's own `UnsafeOnCompleted` therefore does **not** re-capture EC — that is the entire point of the `Unsafe` variant.
- **`SetException` faults the task; it does not throw**. The exception travels into the returned `Task` and is re-raised when the *caller* awaits and calls `GetResult()`. This is the entire mechanism of exception capture in async.

The builder protocol and `ExecutionContext` flow

The builder is the least-discussed and most load-bearing piece. Its method surface — `Create`, `Start`, `SetStateMachine`, `SetResult`, `SetException`, `AwaitOnCompleted`, `AwaitUnsafeOnCompleted`, and the `Task` property — is itself a

duck-typed pattern: any type you name in

`[AsyncMethodBuilder(typeof(...))]` can play, which is how `ValueTask`, `IAsyncEnumerable`, and custom return types like `UniTask` reuse the same compiler transform. Walk the lifecycle:

- **Create()** mints a fresh builder. For `AsyncTaskMethodBuilder<T>` this is nearly free — the actual `Task<T>` is created lazily, only if the method suspends. A method that completes synchronously may avoid a `Task<T>` allocation entirely, but only when the result matches a cached value: the non-generic builder reuses one shared completed `Task` (a `VoidTaskResult` singleton), and `AsyncTaskMethodBuilder<T>` caches only `default(T)` plus a small set of `bool` / `int` values (`AsyncTaskCache`). Completing synchronously with an *arbitrary* `T` (a non-cached value) still allocates one `Task<T>` to carry it. So “no allocation on the sync path” holds for `Task`, for `default(T)`, and for `ValueTask<T>` / `ValueTask` — not for `Task<T>` returning a fresh value.
- **Start(ref sm)** runs `MoveNext` synchronously on the calling thread, but first saves the current `ExecutionContext` and restores it when `MoveNext` returns. That save/restore — not a capture for the continuation — is what stops `AsyncLocal<T>` mutations made *inside* the async method from leaking back into the caller’s context (`ExecutionContext` is copy-on-write, so each suspension segment runs against its own snapshot). The capture that the continuation actually resumes under is taken later, on the first real suspension, when `AwaitUnsafeOnCompleted` boxes the state machine. `AsyncLocal<T>` values set *before* the first `await` are visible to that capture and so flow into continuations.
- **AwaitUnsafeOnCompleted(ref awaiter, ref sm)** is called at each real suspension. It (1) on first call lazily promotes the struct state machine to a heap box and calls `SetStateMachine` so the box holds a builder pointing back at the heap copy (not the about-to-die stack copy), capturing `ExecutionContext` into that box, (2) builds an `Action`

continuation that re-enters the box's `MoveNext` under the captured `ExecutionContext`, and (3) hands that continuation to `awaiter.UnsafeOnCompleted`. The `Unsafe` variant exists precisely because the builder already owns the `ExecutionContext` capture — having the awaiter capture it again would double the cost and break the flow semantics.

- **`SetResult` / `SetException`** transition the underlying `Task` to `RanToCompletion` or `Faulted` / `Canceled`, run any registered continuations, and — for `SetException` — stash the exception via `ExceptionDispatchInfo` so the eventual `GetResult()` can rethrow it with the original stack trace intact.

This is also where the difference between `AwaitOnCompleted` and `AwaitUnsafeOnCompleted` matters for *your* custom awaiters. The builder always captures `ExecutionContext` once (into the box) regardless of which path runs. The difference is the *awaiter*: if your awaiter only implements `INotifyCompletion`, the builder routes through `AwaitOnCompleted`, which hands the continuation to the awaiter's `OnCompleted` — and `OnCompleted` is expected to flow `ExecutionContext` itself, producing a redundant second capture. If your awaiter implements `ICriticalNotifyCompletion`, the builder routes through `AwaitUnsafeOnCompleted` and calls `UnsafeOnCompleted`, which is documented *not* to flow EC, trusting the builder's single capture. Implementing the critical interface is therefore not a micro-optimization curiosity — it changes which builder method runs and removes a redundant capture (and its allocation) on every suspension.

SynchronizationContext, ConfigureAwait, and ConfigureAwaitOptions

`TaskAwaiter`'s `OnCompleted` doesn't blindly queue the continuation to the thread pool. It captures `SynchronizationContext.Current` (falling back to `TaskScheduler.Current`) at the point of suspension and `Post`s the continuation back to it. A `SynchronizationContext`

is an abstraction over “where should this continuation run” — its `Post(callback, state)` method means “schedule this onto my world.” On WPF/WinForms that world is the single UI message-pump thread (`DispatcherSynchronizationContext`), so the continuation marshals back to the UI thread and you can touch controls after an `await` without a manual `Dispatcher.Invoke` . On classic ASP.NET (System.Web) the context (`AspNetSynchronizationContext`) serialized continuations onto the per-request execution flow so `HttpContext.Current` survived an `await`. That capture is exactly what makes `await` feel like it “resumes where I started” — and it is exactly what makes blocking on `async` deadlock, because the one thread the continuation needs is the one you blocked.

ASP.NET Core deliberately installs **no** `SynchronizationContext` . Continuations run on thread-pool threads, request state flows through DI and `IHttpContextAccessor` (an `AsyncLocal` under the hood) rather than a context, and `ConfigureAwait(false)` is consequently a no-op there. It is still good hygiene in shared libraries that might also run under WPF or legacy ASP.NET.

`ConfigureAwait(false)` returns a `ConfiguredTaskAwaitable` whose awaiter does **not** capture the context, so the continuation runs on whatever thread completed the awaited operation (typically a pool thread). In library code that never touches UI/HTTP-context-bound state, `ConfigureAwait(false)` on every `await` is the correct default: it avoids an unnecessary marshal and removes one deadlock vector.

Introduced in **.NET 8 (C# 12 era)**, `ConfigureAwait(ConfigureAwaitOptions)` made the knob a flags enum:

```
[Flags]
public enum ConfigureAwaitOptions
{
    None = 0x0,
    ContinueOnCapturedContext = 0x1, // the "true" you
        used to pass
    SuppressThrowing = 0x2, // await
        completes even on fault/cancel
    ForceYielding = 0x4, // never complete
        synchronously; always suspend
}
```

`ForceYielding` is the composable equivalent of `Task.Yield()` you can OR onto an existing await. `SuppressThrowing` is the interesting one:

```
// Cancel an in-flight background task and wait for it
    to actually unwind,
// without caring whether it ended faulted, canceled,
    or successfully.
_cts.Cancel();
await
    _backgroundTask.ConfigureAwait(ConfigureAwaitOptions.SuppressThrowing);
// Execution continues here regardless of how
    _backgroundTask ended.
// The fault is considered "observed" –
    TaskScheduler.UnobservedTaskException
// will NOT fire for it.
StartReplacement();
```

The deliberate, idiomatic use: you are tearing down and only need the *completion* signal, not the outcome. Because the await observes the exception, you also suppress the finalizer-thread `UnobservedTaskException` event for that task — which is exactly what you want during teardown.

The hard constraint: `SuppressThrowing` is supported only on non-generic `Task`. On `Task<TResult>` it would have to invent a `TResult` to return after swallowing the exception, so at runtime it throws `ArgumentOutOfRangeException`. To catch this at build time, the analyzer **CA2261** (shipped with .NET 8, default severity Warning — raise it to error via `.editorconfig` if you want a hard gate) flags

`someTaskOfT.ConfigureAwait(ConfigureAwaitOptions.SuppressThrowing)` . If you need to suppress on a `Task<T>` , cast to the non-generic base (`((Task)t).ConfigureAwait(...)`) or restructure.

Hot vs cold tasks, exception capture, AggregateException unwrapping

A `Task` returned from an `async` method is **hot**: it is already started (the stub called `builder.Start` before returning), and `MoveNext` has already run synchronously up to the first incomplete `await` . There is no “cold task you must `.Start()` ” in idiomatic `async` — `Task.Start()` / `RunSynchronously` apply only to the legacy `new Task(delegate)` constructor pattern, which you should never use in new code. This hotness has a consequence developers trip over: calling an `async` method *starts the work immediately*, even if you don’t `await` the returned task until later. Side effects up to the first real suspension happen at the call site, not the `await` site.

```
Task<int> t = ComputeAsync(); // runs ComputeAsync's
                             // body up to its first incomplete await – NOW
DoSomethingElse();           // runs concurrently
                             // with whatever ComputeAsync suspended on
int r = await t;              // only the *waiting*
                             // happens here
```

Genuine cold-by-design — work that does not begin until requested — is a property of `IAsyncEnumerable` (the producer doesn’t run until you start enumerating), `Task` -returning *factory delegates* (`Func<Task>`), and `Lazy<Task<T>>` , not of `async` methods themselves. If you want lazy, hand around a delegate, not a task.

Exception flow has two distinct re-raise paths:

```

async Task FaultsAsync() => throw new
    InvalidOperationException("boom");

// Path 1 – await: the captured exception is rethrown
// UNWRAPPED, with stack preserved.
try { await FaultsAsync(); }
catch (InvalidOperationException ex) { /* exactly the
    original */ }

// Path 2 – blocking: .Result / .Wait() / .GetResult()
// on Task wrap in AggregateException.
try { FaultsAsync().Wait(); }
catch (AggregateException agg)
{
    var inner = agg.InnerException; //
    InvalidOperationException
}

```

`await` uses `ExceptionDispatchInfo` to rethrow the *original* exception with its original stack trace, not an `AggregateException`. The wrapping only happens through the synchronous `Task.Wait()` / `Task.Result` surface, which predates `async` and must surface multiple faults (a `WhenAll` can hold many). `ValueTask`'s `.Result` and `TaskAwaiter.GetResult()` both throw unwrapped. Practical rule: never inspect `AggregateException` in `async` code; if you see it, someone blocked. When you genuinely need *all* faults from a fan-out, remember that `await whenAllTask` rethrows only the *first* exception — so keep the task in a variable, `await` it inside a `try`, and in the `catch` read `whenAllTask.Exception!.InnerExceptions` off the now-faulted `Task` to enumerate every fault.

Task vs Task<T> vs ValueTask vs ValueTask<T>
at the type level

Type	Backing	Allocates on async path	Awaits more than once
Task	reference type, one heap object	yes (the Task + boxed SM)	yes
Task<T>	reference type	yes	yes
ValueTask	readonly struct over Task / IValueTaskSource	no, when completed synchronously	no (sum of calls to await is actually once)
ValueTask<T>	readonly struct	no, when sync	no

`ValueTask<T>` is a discriminated union: it holds either a `T` (synchronous result, zero allocation), a `Task<T>` (fell to async), or an `IValueTaskSource<T>` (poolable backing, e.g. `Socket` / `PipeReader`). It exists to remove the `Task<T>` al-

location on the common synchronous-completion case. The cost: a struct with strict single-consumption rules.

```
// CORRECT ValueTask usage: await it exactly once,  
// immediately.  
ValueTask<int> vt = ReadByteAsync();  
int b = await vt;  
  
// WRONG – every one of these is undefined behavior on  
// a non-completed ValueTask:  
ValueTask<int> vt = ReadByteAsync();  
int x = await vt;  
int y = await vt; // double-await: may  
// throw, may return garbage  
bool done = vt.IsCompleted; // reading state after  
// consumption  
await Task.WhenAll(vt.AsTask(), other.AsTask()); //  
// .AsTask() AFTER awaiting is wrong
```

When you must store, await twice, or pass to combinators like `WhenAll`, call `.AsTask()` **once, immediately**, and use that `Task` thereafter. Default to `Task<T>` for public APIs; reach for `ValueTask<T>` only on profiled hot paths (stream readers, channel/pipe primitives, cache getters) where the synchronous-completion ratio is high.

The `IValueTaskSource<T>` backing deserves a mention because it is the reason `ValueTask` is more than a sync-result optimization. When `Socket.ReceiveAsync` or `PipeReader.ReadAsync` does go async, it doesn't allocate a `Task` — it returns a `ValueTask<T>` wrapping a reusable `IValueTaskSource<T>` (typically an `AwaitableSocketAsyncEventArgs` or a pooled `ManualResetValueTaskSourceCore<T>`). That source is reset and reused for the next operation, so a socket pumping millions of reads allocates essentially zero per-read. The single-consumption rule exists *because* of this reuse: read the `ValueTask` twice and you may observe a value belonging to a later, unrelated operation that has since been recycled into the same source object. The struct carries a `short` token that the source validates on `GetResult` to catch some misuse, but it is not a guarantee — treat single-await as inviolable.

For the *output* side, `ValueTask` returned from an `async` method can also pool the state-machine box itself. Opting a method into

`[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))]` (or enabling pooling globally) makes the builder rent the boxed state machine from a per-thread pool instead of allocating a fresh one on each suspension, driving steady-state allocation on a hot async method toward zero. The caveat: pooled boxes are returned to the pool on completion, so any bug that retains a reference to the state machine past completion — or awaits the resulting `ValueTask` twice — corrupts the pool. It is an advanced, profile-driven opt-in, not a default.

async ref locals and unsafe contexts (C# 13)

Before C# 13, you could not declare a `ref` local, a `ref struct` local, or an `unsafe` block inside an `async` or iterator method at all — the compiler rejected it because hoisting a `ref` into a state-machine field is unsound. **Introduced in C# 13 (.NET 9)**, the language lifted that blanket ban: you *may* use `ref` locals, `ref struct` locals (like `Span<T>` / `ReadOnlySpan<T>`), and `unsafe` contexts in async and iterator methods — **provided they do not live across an `await` or `yield`**. The compiler enforces this: the `ref` / `ref struct` must be fully consumed within a single synchronous segment between suspension points, so it never needs hoisting to a field.

This unlocks `Span<T>`-based parsing inside async methods, which was previously impossible without splitting into a synchronous helper:

```

async IEnumerable<long> SumWindowsAsync(
    IEnumerable<byte[]> chunks,
    [EnumeratorCancellation] CancellationToken ct =
        default)
{
    await foreach (byte[] chunk in
        chunks.WithCancellation(ct))
    {
        // C# 13: a ref struct (Span<T>) local inside
        the loop body of an
        // async iterator. Legal because it is created
        and fully consumed
        // between awaits – it never crosses the
        'await foreach' boundary.
        ReadOnlySpan<byte> span = chunk;
        long sum = 0;
        ref readonly byte first = ref span[0]; //
        ref local, also C#13-legal here
        for (int i = 0; i < span.Length; i += 4)
            sum +=
                System Buffers.Binary.BinaryPrimitives.ReadInt
                32LittleEndian(
                    span.Slice(i, 4));

        yield return sum + first; // 'span'/'first'
        are dead before the implicit
        // await of the next
        MoveNextAsync – no hoisting
    }
}

```

The moment you try to keep that `Span<byte>` alive past the `yield return / await`, you hit **CS4007** (“cannot be preserved across an ‘await’ or ‘yield’ boundary”); using a `ref / ref struct / unsafe` feature under a language version below 13 instead reports a **CS9202**-style “feature is not available in C# 12.0” error. The C# 13 feature is precisely “synchronous spans in async bodies,” not “spans that survive suspension” — the latter remains impossible because a `ref struct` cannot become a heap field. The `ReadOnlySpan<byte> span = chunk;` assignment already compiled before C# 14: `byte[] → ReadOnlySpan<byte>` has been an implicit conversion since spans shipped, via the user-defined implicit operator on `ReadOnlySpan<T>`. What **C# 14 (.NET 10)** adds is making that conversion a *first-class, built-in standard conversion* — so it now participates in overload resolution, generic type inference,

and extension-method receivers (e.g. an extension on `ReadOnlySpan<T>` becomes applicable to a `T[]`). It is an overload-resolution and inference change, not “now you can omit a cast you previously needed.”

Deadlocks from blocking on async — and the fix

This is the canonical async crash, and it is purely about the captured `SynchronizationContext`.

```
// WPF / WinForms button handler – runs on the single
// UI thread, which has a
// SynchronizationContext that marshals continuations
// BACK to itself.
void OnClick(object sender, RoutedEventArgs e)
{
    // Block the UI thread on an async result:
    int value = LoadAsync().Result;    // <-- DEADLOCK
    Display(value);
}

async Task<int> LoadAsync()
{
    await Task.Delay(100);              // captures the
    // UI SynchronizationContext
    return 42;                          // continuation
    // must run on the UI thread...
}
```

The deadlock mechanism, step by step: `LoadAsync` suspends at `await Task.Delay`, capturing the UI context. The UI thread, meanwhile, is blocked inside `.Result` waiting for the task to complete. When the delay fires, the continuation (`return 42` and the `SetResult` that completes the task) is `Post`ed to the UI context — but the UI thread is frozen on `.Result` and will never pump that message. The task can’t complete until the continuation runs; the continuation can’t run until the thread is free; the thread can’t free until the task completes. Classic cycle.

Two fixes, in order of preference:

```

// FIX 1 (best): don't block. Make the handler async
// and await.
async void OnClick(object sender, RoutedEventArgs e)
    // async void IS correct for handlers
{
    int value = await LoadAsync();
    Display(value);
}

// FIX 2 (library-side): break the context capture so
// the continuation
// runs on a pool thread and never needs the frozen UI
// thread.
async Task<int> LoadAsync()
{
    await Task.Delay(100).ConfigureAwait(false); // <-
    // no capture, no deadlock
    return 42;
}

```

`ConfigureAwait(false)` in the library lets the continuation resume on a thread-pool thread, so `SetResult` runs there, the task completes, and the blocked `.Result` unblocks. But note: Fix 2 makes the *caller's* sin survivable; it does not make blocking *good*. The real lesson is **don't block on async** — `async` “all the way” is the rule.

`.Result` / `.Wait()` / `.GetAwaiter().GetResult()` on a context-capturing task from a context-bound thread is the bug. (Modern ASP.NET Core has no `SynchronizationContext`, so this specific deadlock doesn't reproduce there — but blocking still wastes a thread-pool thread and risks pool starvation under load, so the prohibition stands.)

A worked example tying it together

A bounded async producer/consumer over `System.Threading.Channels`, with a deliberate `YieldToThreadPool` hop to offload CPU work, span-based decoding (C# 13), and correct teardown via `SuppressThrowing` (.NET 8). This is the kind of pipeline where every concept above earns its keep.


```

using System.Buffers.Binary;
using System.Threading.Channels;

sealed class FramePipeline : IAsyncDisposable
{
    private readonly Channel<byte[]> _channel =
        Channel.CreateBounded<byte[]>(new
            BoundedChannelOptions(128)
            {
                SingleReader = true, SingleWriter = false,
                FullMode = BoundedChannelFullMode.Wait
            });
    private readonly CancellationTokenSource _cts =
        new();
    private readonly Task _consumer;

    public FramePipeline() => _consumer =
        ConsumeAsync(_cts.Token);

    public ValueTask WriteAsync(byte[] frame) =>
        _channel.Writer.WriteAsync(frame, _cts.Token);
    // ValueTask: usually completes sync

    private async Task ConsumeAsync(CancellationToken
        ct)
    {
        long running = 0;
        // ConfigureAwait(false): this is library
        // plumbing, no context to keep.
        await foreach (byte[] frame in
            _channel.Reader.ReadAllAsync(ct).ConfigureAwait(
                false))
        {
            // Decode synchronously with a Span –
            // legal in an async method since C# 13,
            // because the span is dead before the
            // next await (ReadAllAsync's MoveNext).
            ReadOnlySpan<byte> span = frame;
            // byte[] -> ReadOnlySpan<byte>; in C# 14 a
            // built-in conversion
            int header =
                BinaryPrimitives.ReadInt32LittleEndian(span);
            running += header;

            if (CpuExpensive(header))
            {
                // Get OFF the channel-reader thread
                // for the heavy bit, unconditionally.
                await YieldToThreadPool.Instance;
                running += HeavyHash(frame); // now on
                // a pool thread, context-free
            }
        }
    }
}

```

```

    }
    Console.WriteLine($"drained, total=
{running}");
}

public async ValueTask DisposeAsync()
{
    _channel.Writer.TryComplete();
    _cts.Cancel();
    // Wait for the consumer to unwind WITHOUT
    // caring how it ended
    // (it will throw OperationCanceledException –
    // we don't want it surfaced here).
    await
        _consumer.ConfigureAwait(ConfigAwaitOptions
            .SuppressThrowing);
    _cts.Dispose();
}

private static bool CpuExpensive(int h) => (h &
    0xFFFF) > 0x8000;
private static long HeavyHash(byte[] data) { long
    h = 1469598103934665603; foreach (var b in
    data) h = (h ^ b) * 1099511628211; return h; }
}

```

Why each choice is correct: `WriteAsync` returns `ValueTask` because a bounded channel with capacity almost always accepts synchronously — no `Task` allocation on the common path. The consumer uses `ConfigureAwait(false)` throughout because it owns no UI/request context. The `ReadOnlySpan<byte>` decode lives entirely between awaits — its very *presence* inside an async method is C# 13 legality; the `byte[]` → `ReadOnlySpan<byte>` assignment itself has always compiled (C# 14 only reclassifies it as a built-in conversion). `YieldToThreadPool` deliberately hops off the single reader thread so heavy hashing doesn't block the channel drain. `DisposeAsync` uses `SuppressThrowing` to swallow the expected cancellation fault during teardown — non-generic `Task`, so CA2261 is satisfied.

Pitfalls and gotchas

Blocking on async deadlocks on a captured context.

Covered above. The signature smell is `.Result`, `.Wait()`, or `.GetAwaiter().GetResult()` on a task that came from context-

capturing code, called from a UI thread or any thread with a non-null `SynchronizationContext.Current`. Fix: go async all the way; never block. If a sync-over-async bridge is truly unavoidable (e.g., a `Main` before C# 7.1's async `Main`, or an interface you can't change), offload via `Task.Run(() => Async().GetAwaiter().GetResult())` so the inner async runs on a context-free pool thread — but treat that as a code smell requiring a comment.

async void outside event handlers swallows exceptions and crashes the process.

`async void` uses `AsyncVoidMethodBuilder`, which has no `Task` to fault into. An exception that escapes an `async void` method is rethrown on the captured `SynchronizationContext` (or the thread pool if none), where nothing is awaiting it — so it propagates as an unhandled exception and tears down the process. It also can't be awaited, so callers can't know when it finished or whether it failed.

```
async void FireAndForget()           // BAD outside an
    event handler
{
    await Task.Delay(10);
    throw new InvalidOperationException(); // crashes
    the PROCESS, not catchable by caller
}

// Use async Task and let the caller observe it – or,
// for true fire-and-forget,
// be explicit about discarding while still observing
// faults:
static async Task SafeFireAndForget(Func<Task> work)
{
    try { await work().ConfigureAwait(false); }
    catch (Exception ex) { Log(ex); } // exception has
    somewhere to go
}
_ = SafeFireAndForget(FireAndForgetCore);
```

`async void` is correct exactly once: top-level event handlers (`void OnClick(object, EventArgs)`) whose signature the framework dictates. Everywhere else, return `Task`.

Assuming `await` always yields. A task that's already complete continues `MoveNext` synchronously — no thread hop, no scheduling. A loop of `await`s over a cache that's warm runs entirely synchronously and can starve other queued work on the same thread (and, ironically, can re-enter and overflow the stack on deep synchronous chains). If you have a tight loop awaiting potentially-completed tasks and need fairness, insert `await Task.Yield()` (or `ConfigureAwait(ConfigureAwaitOptions.ForceYielding)`) periodically to force a genuine yield point. Conversely, do **not** sprinkle `Task.Run` to “make it async” — wrapping synchronous work in a task just moves it to a pool thread and adds allocation; it doesn't make the work non-blocking.

Reading the lowered form to debug. When a bug smells like “my `AsyncLocal` value vanished,” “my exception came back wrapped,” or “this `ValueTask` threw on second await,” drop the method into SharpLab and look at the generated `MoveNext`: which locals got hoisted to fields (those survive suspension), where `AwaitUnsafeOnCompleted` is called (those are real yield points where context flows), and whether the builder is the `Task`, `ValueTask`, or `AsyncVoid` variant (that tells you the exception destination). The lowering is the ground truth; the source is the sugar.

Performance notes

The synchronous fast path is genuinely free of heap allocation: struct state machine on the stack, no `Task` box, no closure — *if* every awaited operation completes synchronously and your return type is `ValueTask` / `ValueTask<T>`. The first real suspension costs exactly one box of the state machine plus one `Task` allocation (for `Task`-returning methods).

`ConfigureAwait(false)` saves a `SynchronizationContext.Post` marshal per await in library code — small per call, large in aggregate. Prefer `static` lambdas and the generic `QueueUserWorkItem<TState>` / `UnsafeQueueUserWorkItem<TState>` overloads in custom awaiters to keep continuations clo-

sure-free. Implement `ICriticalNotifyCompletion` so the builder takes the `UnsafeOnCompleted` path and avoids a redundant `ExecutionContext` capture. And measure: the JIT inlines small `MoveNext` bodies and the synchronous fast path aggressively, so micro-optimizing an async method that always suspends on real I/O is usually wasted effort — the I/O dominates. Spend the budget where the synchronous-completion ratio is high (channels, pipes, buffered streams), which is exactly where `ValueTask` pays off.

18 · ValueTask, Pooling, and Async Performance

`Task` is a remarkably good general-purpose async primitive, but it is an allocation, and on a hot path that allocation can dominate. This chapter is about the tools the BCL gives you to drive async allocation toward zero — `ValueTask<T>`, `IValueTaskSource`, pooled method builders, and cached completed tasks — and, just as importantly, about the strict consumption rules that make those tools safe and the measurements that tell you whether they were worth it.

The cost model: what an `async Task` method actually allocates

Before reaching for `ValueTask`, understand what you are trying to avoid. A C# `async` method compiles to a state machine. When the method completes **synchronously** — every `await` inside it observed an already-completed awaitable — the compiler-generated `AsyncTaskMethodBuilder` never boxes the state machine onto the heap; for `Task` it hands back a cached singleton (`Task.CompletedTask`) or, for `Task<T>`, a cached task when the value is cheap to cache (e.g. small integers, booleans). When the method **suspends** — an `await` hits an incomplete awaitable — the builder lifts the state machine to the heap as an `AsyncStateMachineBox<T>`. Since .NET Core 2.1 that box *derives from* `Task<T>`, so the box and the returned task are the **same object**: a single heap allocation on the suspending path (not the two — box plus separate `Task` — that older runtimes incurred). On the synchronous-but-uncacheable path you still pay *one* allocation: the `Task<T>` that carries the result.

The synchronous-but-uncacheable case is the one `ValueTask<T>` targets. Consider a cache lookup that hits 95% of the time: you pay a `Task<T>` allocation on every hit for a result you computed without ever yielding. `ValueTask<T>` lets that hit return without touching the heap.

What `ValueTask<T>` is, and what it costs

`ValueTask<T>` is a readonly struct that wraps **one of three** things in an `object` field plus a couple of small fields:

- a `T` result directly (synchronous completion),
- a `Task<T>` (the operation went async and fell back to a task), or
- an `IValueTaskSource<T>` plus a `short` token (a pooled/reusable backing source).

```
// Conceptual shape – the real struct lives in
// System.Threading.Tasks.
public readonly struct ValueTask<TResult>
{
    private readonly object? _obj;    // null => use
    // _result; else Task<T> or IValueTaskSource<T>
    private readonly TResult? _result;
    private readonly short _token;
    private readonly bool _continueOnCapturedContext;
}
```

The win: when `_obj` is null, the whole thing is a value on the stack — no GC pressure, no indirection. The cost: a

`ValueTask<T>` is larger than a `Task<T>` reference (a pointer), so passing it around copies more bytes; and because it can wrap a *pooled, reusable* source, it carries fragile lifetime rules that `Task` does not. `Task` is freely shareable, awaitable any number of times, by any number of consumers. `ValueTask<T>` is not. That single difference is the source of every gotcha in this chapter.

Returning `ValueTask` / `ValueTask<T>` is a micro-optimization with sharp edges. The .NET team's own guidance: default to `Task` / `Task<T>` on public APIs and reach for `ValueTask` only where profiling shows the allocation matters and you control (or trust) the call site to follow the rules.

The three consumption rules — memorize these

A `ValueTask` / `ValueTask<T>` may wrap a backing object that is **recycled** the moment the result is observed. Therefore:

1. **Await it at most once.** A second await reads a source that may have been reset and handed to a different operation. The result is silently wrong or corrupt.
2. **Never await it concurrently.** Two awaits in flight against one pooled source race on its state.
3. **Never block on it** via `.Result` / `.GetAwaiter().GetResult()` unless `IsCompleted` is true. A pooled source is not designed to be synchronously waited on; `Task` is.

The single safe consumption pattern is “await it immediately, exactly once”:

```
ValueTask<int> vt = cache.GetAsync(key);  
int value = await vt;           // good: awaited once,  
                               immediately
```

If you need to do anything else — await later, await twice, store it, pass it to multiple consumers, `WhenAll` it — **convert to Task first** with `.AsTask()` and use that:

```
Task<int> t = cache.GetAsync(key).AsTask(); //  
      .AsTask() consumes the ValueTask exactly once  
int a = await t;                          // Task is  
      freely re-awaitable  
int b = await t;                          // fine
```

`.AsTask()` is itself a one-time consumption of the `ValueTask`, so call it once and use the resulting `Task` thereafter. The only allocation-free way to peek without consuming is `IsCompleted` / `IsCompletedSuccessfully`, which are safe to read.

Worked example: a cache returning `ValueTask<T>`

Here is the canonical good use of `ValueTask<T>` : a cache whose hit path completes synchronously and allocates nothing, while the miss path falls back to a real `Task` .

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

public sealed class WeatherCache(IWeatherClient
    client)
{
    private readonly ConcurrentDictionary<string,
        decimal> _hot = new();

    public ValueTask<decimal>
        GetTemperatureAsync(string city,
            CancellationToken ct = default)
    {
        // Fast path: synchronous completion, ZERO
        // heap allocation.
        if (_hot.TryGetValue(city, out decimal
            cached))
            return new ValueTask<decimal>(cached);

        // Slow path: actually await I/O. Wrap the
        // resulting Task.
        return new ValueTask<decimal>
            (FetchAndStoreAsync(city, ct));
    }

    private async Task<decimal>
        FetchAndStoreAsync(string city,
            CancellationToken ct)
    {
        decimal value = await
            client.GetTemperatureAsync(city, ct); //
            // suspends → Task path
        _hot[city] = value;
        return value;
    }
}

public interface IWeatherClient
{
    Task<decimal> GetTemperatureAsync(string city,
        CancellationToken ct);
}
```

Two design points worth dwelling on. First, `GetTemperatureAsync` is **not** itself `async`. It is a plain method returning `ValueTask<decimal>`, so on a hit there is no state machine and no builder at all — just `new ValueTask<decimal>(cached)`, a struct construction. The `async` keyword is pushed down into `FetchAndStoreAsync`, which only runs on a miss. This split — synchronous wrapper, async core — is the idiom for squeezing every allocation out of the hot path. Second, the miss path wraps a `Task<decimal>`, not a pooled source, so the returned `ValueTask` is as safe as a `Task`; the only constraint the caller inherits is “await once.”

The primary constructor (`WeatherCache(IWeatherClient client)`) is the C# 12 form, introduced in .NET 8; nothing here requires C# 14, which is exactly the point — `ValueTask` patterns are stable and boring, and that is a virtue.

Caching completed results: `Task.CompletedTask`, `Task.FromResult`, and `ValueTask` defaults

The cheapest async result is one you never allocate. The runtime already caches some for you:

- `Task.CompletedTask` — a singleton completed `Task`. Return it from `async`-free methods that have nothing to await.
- `Task.FromResult(value)` — since .NET 6 it shares the **same** `AsyncTaskCache` that `async` methods use, so it returns a cached singleton for the values that cache covers (`Task<bool>` for `true` / `false`, `Task<int>` for the small range `-1...8`, and `null` / `default` references). For *any other* value — a `decimal`, a string, an `int` outside `-1...8` — it allocates a fresh `Task<T>` on every call. That uncacheable-value case is the line you most often want to delete on a hot path.
- `ValueTask.CompletedTask` and `default(ValueTask)` — both represent a successfully completed `ValueTask` with **no allocation** (the property is literally `default`).

`default(ValueTask<T>)` is a completed `ValueTask<T>` whose result is `default(T)`.

```
// Hot path that returns void-ish completion: prefer the cached singleton.
public Task FlushAsync() => _buffer.IsEmpty ?
    Task.CompletedTask : FlushCoreAsync();

// default(ValueTask) is a free, completed ValueTask – no Task.CompletedTask indirection.
public ValueTask DisposeAsync() => _disposed ? default
    : DisposeCoreAsync();
```

Replacing `Task.FromResult` on a hot path — measured

`Task.FromResult` in a tight loop is a steady allocation drip. Swapping it for `ValueTask<T>`'s value constructor removes it entirely on the synchronous path. Here is a BenchmarkDotNet harness that proves it.

```

using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System.Threading.Tasks;

[MemoryDiagnoser]           // reports Allocated
                             bytes/op
public class FromResultBench
{
    private const int Iterations = 1_000;

    [Benchmark(Baseline = true)]
    public async Task<long> TaskFromResult()
    {
        long sum = 0;
        for (int i = 0; i < Iterations; i++)
            sum += await Lookup_Task(i); // each
            // await consumes a freshly allocated Task<int>
        return sum;
    }

    [Benchmark]
    public async Task<long> ValueTaskValue()
    {
        long sum = 0;
        for (int i = 0; i < Iterations; i++)
            sum += await Lookup_ValueTask(i); // no
            // allocation: synchronous completion
        return sum;
    }

    // Always-synchronous lookups to isolate the
    // allocation difference.
    // Offset by 1000 so every value lands OUTSIDE
    // Task.FromResult's cached -1..8
    // int range – otherwise the cache hides part of
    // the allocation we're measuring.
    private static Task<int> Lookup_Task(int i) =>
        Task.FromResult(i + 1000);
    private static ValueTask<int> Lookup_ValueTask(int
        i) => new(i + 1000);
}

// public static class Program { public static void
//     Main() => BenchmarkRunner.Run<FromResultBench>
//     (); }

```

Representative output on .NET 10:

Method	Mean	Ratio	Allocated	
Alloc Ratio				
-----:	-----:	-----:	-----:	---
-----:				
TaskFromResult	12.4 μs	1.00	72,000 B	
1.00				
ValueTaskValue	4.1 μs	0.33	0 B	
0.00				

The numbers will vary, but the shape is invariant: with values forced outside the cached `-1..8` range, `Task.FromResult` allocates one `Task<int>` per call (`~72 bytes × Iterations` on 64-bit), and the `ValueTask<int>` value path allocates nothing. Note the caveat baked into the benchmark: this only holds because the lookup *completes synchronously*. If `Lookup_ValueTask` had to await real I/O, both would allocate, and `ValueTask` would buy you only the wrapper savings — which is why you measure the realistic hit/miss ratio, not the best case.

`IValueTaskSource<T>` : zero-allocation async on the asynchronous path

Wrapping a `Task` makes the async path cost a `Task` allocation. To get to *zero* allocations even when the operation genuinely suspends, you implement `IValueTaskSource<T>` and **reuse** the backing object across operations. This is how `Socket`, `Channel<T>`, and `PipeReader` achieve allocation-free `ReadAsync` under load.

You almost never implement the three interface methods by hand. Instead you embed

`ManualResetValueTaskSourceCore<T>` (introduced in .NET Core 3.0) and delegate to it; it correctly handles continuation scheduling, `ExecutionContext` / `SynchronizationContext` flow, token versioning, and the completed-vs-pending race.

```

using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Sources;

/// A single reusable async operation. One instance
/// backs many sequential awaits,
/// allocating nothing per operation. NOT thread-safe
/// for concurrent operations -
/// one pending operation at a time, which is the
/// IValueTaskSource contract.
public sealed class ReusableTimerOp :
    IValueTaskSource<bool>
{
    private ManualResetValueTaskSourceCore<bool>
        _core; // mutable struct: never make it
        readonly
    private Timer? _timer;

    public ReusableTimerOp()
    {
        // Pool continuations onto the thread pool
        unless the awaiter opts into context capture.
        _core.RunContinuationsAsynchronously = true;
        _timer = new Timer(static s =>
            ((ReusableTimerOp)s!).OnTimer(), this,
                Timeout.Infinite,
                Timeout.Infinite);
    }

    // Begin one operation. The returned ValueTask
    MUST be awaited exactly once.
    public ValueTask<bool> WaitAsync(TimeSpan delay)
    {
        _core.Reset(); // bump the token; recycle the source
        _timer!.Change(delay,
            Timeout.InfiniteTimeSpan);
        return new ValueTask<bool>(this,
            _core.Version); // (source, token) - the
            pooled form
    }

    private void OnTimer() => _core.SetResult(true);
        // completes the pending ValueTask

    // --- IValueTaskSource<bool> plumbing: pure
    delegation to the core ---
    public bool GetResult(short token) =>
        _core.GetResult(token);
    public ValueTaskSourceStatus GetStatus(short
        token) => _core.GetStatus(token);

```

```

public void OnCompleted(short token,
    Action<object?> continuation,
    object? state,
    ValueTaskSourceOnCompletedFlags flags)
=> _core.OnCompleted(continuation, state,
    token, flags);
}

```

Usage — note the token returned by `_core.Version` is what protects you: each `Reset()` increments it, so an await against a stale token throws instead of silently reading a recycled result.

```

var op = new ReusableTimerOp();
for (int i = 0; i < 1_000_000; i++)
{
    bool fired = await
        op.WaitAsync(TimeSpan.FromMilliseconds(1)); //
        zero alloc per iteration
    // ... one operation completes before the next
    // WaitAsync — never overlap them.
}

```

Four rules govern a hand-rolled source, and the core enforces the first three for you:

- **Token versioning.** Always pass `_core.Version` into the `ValueTask` and back into `GetResult` / `GetStatus` / `OnCompleted`. A mismatched token means the consumer reused or double-awaited; the core throws.
- **`Reset()` before each new operation**, after the previous result has been consumed. Resetting while an await is pending is a bug.
- **One operation in flight at a time.** `IValueTaskSource` is explicitly single-consumer; pooling a source across overlapping operations corrupts state.
- **Don't make `_core` a `readonly` field.** It is a mutable struct; a `readonly` field would defensively copy on every member call, so mutations to `Version` / state would be lost. (The compiler will not warn you.)

This is genuinely difficult code to get right by hand, which is why the next section exists: the runtime ships a pooled builder that gives you most of the benefit for one attribute.

`[AsyncMethodBuilder]` and

`PoolingAsyncValueTaskMethodBuilder<T>`

The `[AsyncMethodBuilder(typeof(...))]` attribute — usable on individual methods since C# 10 (.NET 6); previously it could target only a type — lets you override the state-machine builder for one `async` method without changing its return type. Pair it with `PoolingAsyncValueTaskMethodBuilder` /

`PoolingAsyncValueTaskMethodBuilder<T>`

(`System.Runtime.CompilerServices`) and the runtime **pools the boxed state machine and the backing**

`IValueTaskSource` , so a method that genuinely suspends can complete with *no per-call heap allocation* after warm-up. The pooling machinery first shipped in .NET 5 behind a process-wide environment-variable experiment; it became the *public, per-method opt-in* builder types in .NET 6, once C# 10's method-level `[AsyncMethodBuilder]` override existed to point at them.


```

using System.Buffers.Binary;           //
    BinaryPrimitives
using System.IO;                       // Stream
using System.Runtime.CompilerServices; //
    AsyncMethodBuilder,
    PoolingAsyncValueTaskMethodBuilder
using System.Threading;                //
    CancellationToken
using System.Threading.Tasks;          // ValueTask<T>

public sealed class FrameReader(Stream stream)
{
    private readonly byte[] _header = new byte[4];

    // The attribute overrides only THIS method's
    // builder. The return type stays ValueTask<int>.

    [AsyncMethodBuilder(typeof(PoolingAsyncValueTa
skMethodBuilder<>))]
    public async ValueTask<int>
        ReadFrameLengthAsync(CancellationToken ct)
    {
        int read = 0;
        while (read < 4)
        {
            int n = await
stream.ReadAsync(_header.AsMemory(read), ct);
            // suspends under load
            if (n == 0) return -1; // EOF
            read += n;
        }
        return
BinaryPrimitives.ReadInt32LittleEndian(_header
);
    }
}

```

What the override buys you, and what it costs:

- **Benefit:** when this method awaits an incomplete awaitable, a pooled box — a single object that holds the state machine and implements `IValueTaskSource<int>` — is rented from a per-type pool and returned when the result is consumed. A hot read loop that suspends millions of times allocates essentially nothing steady-state.
- **Cost #1 — pool management overhead.**
Renting/returning has CPU cost and lock contention paths. On methods that mostly complete synchronously, the default

builder is already allocation-free and the pooling machinery is pure overhead. *Apply this only to methods that actually suspend often.*

- **Cost #2 — correctness is non-negotiable.** A pooled `ValueTask<T>` is returned to its pool the instant it is consumed. Double-awaiting, storing, or blocking on one of these now corrupts a *shared* pooled source, not just your own result. Every consumption rule from earlier becomes a hard invariant.
- **Cost #3 — it's surgical, not global.** There was once a global `DOTNET_SYSTEM_THREADING_POOLASYNCTASKS` knob; it was removed precisely because blanket pooling regressed as often as it helped. The supported mechanism is the per-method attribute, applied where end-to-end benchmarks show a win.

Benchmark the attributed method against the plain one under a *suspending* workload (the synchronous path won't show the difference):

```

// PlainReader is FrameReader's twin WITHOUT the
// [AsyncMethodBuilder] attribute,
// so it uses the default
// AsyncValueTaskMethodBuilder<int> (allocates on
// suspend).
public sealed class PlainReader(Stream stream)
{
    private readonly byte[] _header = new byte[4];

    public async ValueTask<int>
        ReadFrameLengthAsync(CancellationToken ct)
    {
        int read = 0;
        while (read < 4)
        {
            int n = await
                stream.ReadAsync(_header.AsMemory(read), ct);
            if (n == 0) return -1;
            read += n;
        }
        return
            BinaryPrimitives.ReadInt32LittleEndian(_header
            );
    }
}

[MemoryDiagnoser]
public class PoolingBench
{
    private readonly FrameReader _pooled =
        new(MakeStream());
    private readonly PlainReader _plain =
        new(MakeStream());

    [Benchmark(Baseline = true)]
    public async ValueTask<int> Plain() => await
        _plain.ReadFrameLengthAsync(default);

    [Benchmark]
    public async ValueTask<int> Pooled() => await
        _pooled.ReadFrameLengthAsync(default);

    // MakeStream() returns a stream whose ReadAsync
    // ALWAYS completes asynchronously
    // (a custom test double), forcing the suspend
    // path so the pooling difference shows.
    // A real DelayStream/SlowStream overrides
    // ReadAsync to yield via Task.Yield()/a timer.
    private static Stream MakeStream() => new
        SlowStream();
}

```

Expect the pooled variant to report 0 B (or near it) allocated per op. The plain variant's default

`AsyncValueTaskMethodBuilder<int>` is *already* allocation-free on the synchronous path; the difference appears only on suspend, where it boxes the state machine onto the heap as an `AsyncStateMachineBox<int>` that derives from `Task<int>` (so the box doubles as the returned `Task`-backed `ValueTask` — one heap object, not two). The pooled builder rents that box from a per-type pool and returns it when the result is consumed, driving steady-state allocation to zero. Expect the pooled variant's *mean time* to be slightly higher or roughly equal — you are trading a little CPU for a lot less GC. Whether that trade wins depends on your allocation budget and GC pressure, which is exactly why this is a measured decision, not a default.

The runtime-async preview (.NET 10) — context, not a recommendation

.NET 10 ships an **experimental, opt-in** “runtime async” implementation that moves suspension out of the compiler-generated state machine and into the runtime: instead of emitting a state machine, the compiler emits a call the runtime intercepts (`AsyncHelpers.Await` -style), suspending and resuming with only the live state preserved. Early measurements show meaningful allocation reductions (the runtime team has cited drops on the order of 60% — e.g. ~360 B → ~144 B — on representative chains) and better performance on deep async call trees.

In .NET 10 this is behind a runtime feature flag and is **not** production-ready; the supported, on-by-default story lands in .NET 11. The practical takeaway for a .NET 10 codebase: the manual techniques in this chapter — `ValueTask` fast paths, pooled sources, `PoolingAsyncValueTaskMethodBuilder` — remain the correct tools today, and runtime-async will eventually subsume much of the *allocation* motivation (though not the *re-awaitability* semantics) of `ValueTask`. Write your hot paths cleanly now; don't contort code around an experiment.

Buffering and backpressure in async pipelines

Allocation per `await` is only one axis of async performance. The other is **backpressure** — bounding in-flight work so a fast producer can't exhaust memory feeding a slow consumer.

`System.Threading.Channels` is the idiomatic primitive, and a *bounded* channel returns a `ValueTask` from `WriteAsync` that completes synchronously when there's room and suspends (allocation-free, via its internal `IValueTaskSource`) when the buffer is full.

```
using System.Threading.Channels;

var channel = Channel.CreateBounded<Frame>(new
    BoundedChannelOptions(capacity: 1024)
{
    FullMode = BoundedChannelFullMode.Wait, //
        producer awaits when full → backpressure
    SingleReader = true, // lets
        the channel skip reader-side synchronization
    SingleWriter = false,
});

// Producer: WriteAsync returns ValueTask — await it
// once, right here.
async Task ProduceAsync(ChannelWriter<Frame> writer,
    IEnumerable<Frame> source,
    CancellationToken ct)
{
    await foreach (Frame f in
        source.WithCancellation(ct))
        await writer.WriteAsync(f, ct); //
        suspends (no alloc) when the buffer is full
    writer.Complete();
}

// Consumer: ReadAllAsync yields a ValueTask-backed
// async stream.
async Task ConsumeAsync(ChannelReader<Frame> reader,
    CancellationToken ct)
{
    await foreach (Frame f in reader.ReadAllAsync(ct))
        Process(f);
}
```

Two performance notes. `SingleReader = true / SingleWriter = true` are not safety toggles — they are *optimization* hints that

let the channel elide synchronization on the declared side; set them honestly. And the `WriteAsync` / `ReadAsync` results are `ValueTask`s, so the same “await once” rule applies inside your pipeline loops: never stash them, never `WhenAll` them without `.AsTask()`.

Measuring async allocation — BenchmarkDotNet and ETW

You cannot reason about async allocation from source alone; the synchronous-completion path is invisible in the code and decisive in the profile. Two complementary tools:

BenchmarkDotNet with `MemoryDiagnoser` gives you allocated-bytes-per-operation and lets you A/B two return-type choices. It is the right tool for micro-decisions: “does `ValueTask` actually remove the allocation here?” Crucially, drive it with a workload that matches your real hit/miss ratio — benchmarking a 100%-synchronous path will oversell `ValueTask`, and a 100%-async path will undersell the value constructor.

`dotnet-trace` / **ETW (`dotnet-counters`)** measures the whole process under realistic load. Watch `GC Heap Size`, `Gen 0 GC Count`, and `Allocation Rate`. A successful `ValueTask`/pooling change shows up as a flatter Gen 0 collection cadence under sustained load — the systemic signal that micro-benchmarks can’t show.

```
# Sample allocation behavior of a running service
under load:
dotnet-counters monitor -n MyService --counters
    System.Runtime[alloc-rate,gen-0-gc-count]

# Capture a GC/allocation trace for offline analysis
(PerfView / Speedscope):
dotnet-trace collect -n MyService --providers
    Microsoft-DotNETCore-SampleProfiler,Microsoft-
    Windows-DotNETRuntime:0x1:5
```

The discipline: change one return type, measure both the micro-benchmark *and* the allocation rate under load. If neither moves

meaningfully, revert to `Task` — you’ve added consumption hazards for nothing.

Pitfall: awaiting the same `ValueTask` twice — corruption, demonstrated

This is the canonical misuse. With a pooled source it produces silent data corruption; the BCL doesn’t always throw, because detecting every double-await would cost the performance the type exists to provide.

```
// ❌ BROKEN. Do not do this.
public async Task DoubleAwaitBug(ReusableTimerOp op)
{
    ValueTask<bool> vt =
        op.WaitAsync(TimeSpan.FromMilliseconds(5));

    bool first = await vt;    // consumes the source;
                             // the source may now Reset() and be reused
    bool second = await vt;  // reads a RECYCLED
                             // source via a STALE token → wrong result or
                             // throw
    // 'second' is meaningless; if the source was
    // handed to another operation, you also
    // corrupted THAT operation's state.
}
```

The hand-rolled `ManualResetValueTaskSourceCore` token check turns the second await into an `InvalidOperationException` — its `ValidateToken` sees a token that no longer matches `_version` and throws (the message is the generic “An attempt was made to transition a task to a final state when it had already completed”) — if you wired the token through correctly. With a `Task`-backed `ValueTask`, the second await happens to work, which is worse: it trains the habit, then breaks the day someone swaps in a pooled source. Treat every `ValueTask` as if it were pooled.

```
// ✅ CORRECT – await exactly once...
public async Task SingleAwait(ReusableTimerOp op)
{
    bool fired = await
        op.WaitAsync(TimeSpan.FromMilliseconds(5));
}

// ✅ CORRECT – ...or convert to Task FIRST when you
// need Task semantics.
public async Task NeedTaskSemantics(WeatherCache
    cache, string city)
{
    Task<decimal> t =
        cache.GetTemperatureAsync(city).AsTask(); //
        one-time consumption
    decimal a = await t;    // re-awaitable
    decimal b = await t;    // safe – it's a Task now
    await Task.WhenAll(t, SomethingElseAsync()); //
        safe – Task composes; raw ValueTask does not
}
```

Pitfall: blocking on a not-yet-completed

ValueTask

`.Result` and `.GetAwaiter().GetResult()` on a `ValueTask` are only valid when `IsCompleted` is true. A `Task` spins up blocking-wait machinery; a pooled `ValueTask` source has none and is free to throw, deadlock, or return garbage.

```
// ❌ Blocks on a possibly-pending pooled source –
// undefined behavior.
int bad = cache.GetTemperatureAsync("Brno").Result;

// ✅ Only read synchronously after confirming
// completion; otherwise await.
ValueTask<decimal> vt =
    cache.GetTemperatureAsync("Brno");
decimal temp = vt.IsCompletedSuccessfully ? vt.Result
    : await vt;
```

That `IsCompletedSuccessfully ? vt.Result : await vt` idiom is the *one* legitimate use of `.Result` on a `ValueTask`, and it's how performance-sensitive BCL call sites avoid an `await` state machine on the synchronous path while staying correct on the async path.

Pitfall: `ValueTask` everywhere by default

The most common real-world mistake isn't corruption — it's reflexively returning `ValueTask` from every async API “for performance.” This taxes every caller with the await-once rule, blocks them from `WhenAll` / `WhenAny` without `.AsTask()` (which re-introduces the allocation you were avoiding, *plus* a `ValueTask` wrapper), and complicates mocking and testing — all for an allocation that, on a method that usually goes async anyway, you never actually saved.

Choosing the return type for a public async API

A decision procedure that holds up in code review:

- **Public/library API, general use** → `Task` / `Task<T>` .
Maximally composable, re-awaitable, the least-surprise default. The allocation is real but usually negligible against the I/O it represents.
- **Nothing to await; completes synchronously** → **cached completion.** `Task.CompletedTask` , `default(ValueTask)` , or a method that isn't `async` at all and returns `new ValueTask<T>(value)` .
- **Hot path, frequently synchronous completion, un-cacheable result** → `ValueTask<T>` . The cache-hit scenario. Measure the hit ratio; the win scales with it.
- **Hot path, frequently *asynchronous*, allocation-critical** → `IValueTaskSource<T>` or `[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))]` . Sockets, channels, pipes, framing readers. Benchmark end-to-end; accept the strict lifetime rules.
- **When in doubt** → `Task` . You can tighten a `Task` API to `ValueTask` later behind the same `await` syntax with no caller breakage on the happy path; loosening a misused `ValueTask` after corruption ships is far worse.

The throughline: `ValueTask` and pooled sources are precision instruments for paths you have *measured*. The synchronous-completion fast path is where `ValueTask<T>` earns its keep;

pooled `IValueTaskSource` and `PoolingAsyncValueTaskMethodBuilder` extend the zero-allocation guarantee onto the genuinely asynchronous path; and the consumption rules — await once, never block on a pending one, never reuse — are the price of admission. Honor the rules, measure the benefit, and default to `Task` everywhere you haven't proven otherwise.

19 · Async Streams, Channels, and Cancellation

Async streams (`IAsyncEnumerable<T>`) and `System.Threading.Channels` are the two pillars of pull-based and push-based asynchronous data flow in modern .NET. This chapter dissects how the compiler turns an async iterator into a state machine, how cancellation tokens actually reach that state machine, how bounded channels give you backpressure for free, and how to shut a pipeline down without losing data or swallowing the fact that someone cancelled. The APIs are stable since .NET Core 3.0, but .NET 10 finally folds LINQ-over-`IAsyncEnumerable<T>` into the BCL, which changes how you compose these pipelines.

Async iterators and `await foreach`

An *async iterator* is a method that returns `IAsyncEnumerable<T>` (or `IAsyncEnumerator<T>`) and contains at least one `yield return` / `yield break` together with at least one `await` . The compiler rewrites it into a struct-based state machine implementing `IAsyncStateMachine` plus `IAsyncEnumerator<T>` , hoisting locals into fields and turning every `await` and every `yield` into a resumption point. Unlike a synchronous iterator, the state machine also implements `IAsyncDisposable` , because cleanup (`finally` blocks, `using`) may itself need to `await` .

The consuming side is `await foreach` . It is pure syntactic sugar over the *async enumerator pattern*: the compiler calls `GetAsyncEnumerator(CancellationToken)` , then loops `while (await e.MoveNextAsync())` reading `e.Current` , and finally `await e.DisposeAsync()` in a `finally` . The pattern has been duck-typed since C# 8 (when async streams shipped) — any type exposing a matching instance `GetAsyncEnumerator` works without implementing `IAsyncEnumerable<T>` ; C# 9 extended that to allow the `GetAsyncEnumerator` to be supplied as an ex-

ension method. The canonical shape, though, is

`IEnumerable<T>`.

```
using System.Net.Http.Json;

// An async iterator that pages through a REST API,
// yielding items one at a time.
// The caller sees a flat stream; paging is an
// implementation detail.
public sealed class GitHubClient(HttpClient http)
{
    public async IEnumerable<Repo>
        GetReposAsync(string org)
    {
        int page = 1;
        while (true)
        {
            // Each await suspends the state machine;
            // no thread is blocked while
            // the HTTP request is in flight.
            var batch = await
                http.GetFromJsonAsync<Repo[]>(
                    $"orgs/{org}/repos?per_page=100&page=
                    {page}")
                ?? []; // collection expression (C# 12)

            if (batch.Length == 0)
                yield break; // terminates the stream cleanly

            foreach (var repo in batch)
                yield return repo; // surfaces one element to the consumer

            page++;
        }
    }
}

public sealed record Repo(long Id, string Name, int
    StargazersCount);
```

Three properties of the rewrite matter for performance. First, the state machine is a `struct` boxed only once (on first `MoveNextAsync` when an `await` actually suspends); a stream that never genuinely yields control allocates nothing on the fast path. Second, laziness is total: no HTTP call happens until the

consumer pulls the first element, and the second page is never fetched if the consumer stops early. Third, back-pressure is structural — the producer cannot run ahead of the consumer because it is the consumer's `MoveNextAsync` that drives it. That last point is the key contrast with channels later in this chapter.

A long-standing restriction was lifted recently: **C# 13 (.NET 9) allows `ref locals`, `ref struct locals`, and `unsafe blocks` inside `async methods and iterators`**, as long as they do not stay alive across an `await` or `yield` suspension point. Before C# 13 you could not so much as declare a `Span<T>` local in an `async iterator`. Now you can use spans for the synchronous slices of work between awaits — the compiler enforces that the span does not survive a suspension (which would be unsound, since the state machine's hoisted fields cannot hold a `ref struct`). This makes it practical to do zero-allocation parsing inside an `async iterator`'s per-element path:

```
public async IAsyncEnumerable<int>
    ParseLengthsAsync(IAsyncEnumerable<string>
        lines)
{
    await foreach (var line in lines)
    {
        // C# 13: Span<char> local is legal here
        // because it never crosses an await.
        ReadOnlySpan<char> span =
            line.AsSpan().Trim();
        int len = span.Length; // fully
        // consumed before the next yield
        yield return len;
    }
}
```

Flowing a `CancellationToken` into the iterator body

Here is the single most misunderstood part of `async streams`.

`IAsyncEnumerable<T>.GetAsyncEnumerator` takes a `CancellationToken`, and `await foreach (... .WithCancellation(token))` passes a token to it. But your iterator *method* has its own parameters, fixed at the moment the it-

erator object is constructed — long before the consumer attaches a token in `GetAsyncEnumerator`. So which token wins?

The answer is the `[EnumeratorCancellation]` attribute (introduced with async streams in C# 8). Decorate the iterator's `CancellationTokens` parameter with it, and the compiler wires that parameter to whatever token the consumer later supplies via `WithCancellation`. The generated forwarding is smart about it: if the call-time argument was `default` (`CancellationTokens.None`), the parameter simply becomes the `WithCancellation` token; if both are the *same* token, it is used directly; and only if the producer was called with a *distinct* non-default token does the generated enumerator build a `CreateLinkedTokenSource` of the two so either one can cancel. (That linked source is disposed by the enumerator's `DisposeAsync`, so you don't leak it.)

```
using System.Runtime.CompilerServices;

public async IAsyncEnumerable<Repo> GetReposAsync(
    string org,
    [EnumeratorCancellation] CancellationToken ct =
        default) // <-- the magic
{
    int page = 1;
    while (true)
    {
        ct.ThrowIfCancellationRequested(); // honor
            cancellation between pages

        var batch = await
            http.GetFromJsonAsync<Repo[]>(
                $"orgs/{org}/repos?per_page=100&page=
                {page}", ct) // pass it downstream too
                ?? [];

        if (batch.Length == 0)
            yield break;

        foreach (var repo in batch)
            yield return repo;

        page++;
    }
}
```

Now both of these cancel the loop correctly:

```
// Consumer supplies the token;  
[EnumeratorCancellation] routes it into ct.  
await foreach (var repo in  
    client.GetReposAsync("dotnet").WithCancellation(token))  
    Console.WriteLine(repo.Name);  
  
// Producer supplies the token directly; same  
parameter, also works.  
await foreach (var repo in  
    client.GetReposAsync("dotnet", token))  
    Console.WriteLine(repo.Name);
```

Pitfall — forgetting `[EnumeratorCancellation]`. Without the attribute, the `ct` parameter is bound *only* to the argument passed at call time. A consumer using `.WithCancellation(token)` will see its token silently dropped: `ct` stays default (`CancellationToken.None`), `ThrowIfCancellationRequested` never throws, and the stream runs to completion no matter how hard the caller tries to cancel. There is no compiler error and no warning at the default analyzer level — the code looks correct and behaves as an uncancelable stream. Treat the attribute as mandatory on every `CancellationToken` parameter of an async iterator. (Analyzer `CA2016` will flag missing forwarding of tokens, but the canonical guard is to just always add the attribute.)

`ConfigureAwait` and `WithCancellation` on `await foreach`

`await foreach` has two extension methods that configure the underlying `MoveNextAsync` / `DisposeAsync` awaits: `WithCancellation(CancellationToken)` and `ConfigureAwait(bool)`. Both return a `ConfiguredCancelableAsyncEnumerable<T>` and are chainable in either order.

```
// Library code: don't capture the synchronization
// context, and honor the token.
await foreach (var repo in client
    .GetReposAsync("dotnet")
    .WithCancellation(token)
    .ConfigureAwait(false))
{
    Process(repo);
}
```

`ConfigureAwait(false)` here applies to the *machinery* awaits (each `MoveNextAsync`), exactly as it would for a single `await`. In library code that must not marshal back to a UI or ASP.NET (legacy) synchronization context, use it on every `await` `foreach`. In application code on a context-free runtime (ASP.NET Core, console, worker), it is a no-op and you can omit it. Note that `ConfigureAwait` on the *foreach* does not propagate to `await`s *inside* the iterator body — the producer’s own awaits are configured by the producer, not the consumer. If your iterator awaits without `ConfigureAwait(false)`, that is the producer’s bug to fix, independent of how the consumer iterates.

Chaining

`.WithCancellation(token).ConfigureAwait(false)` is the idiomatic “library-grade” consumption form, and it is the exact shape you should memorize.

Async disposal and `await using`

Because an async iterator’s state machine implements `IDisposable`, `await foreach` always calls `DisposeAsync`, and your iterator’s `finally` blocks run even when the consumer breaks out early or an exception unwinds the loop. This is what makes resource cleanup correct in the face of cancellation: a `try/finally` (or `await using`) around the producing logic releases handles deterministically.


```

public async IEnumerable<string> ReadLinesAsync(
    string path, [EnumeratorCancellation]
    CancellationToken ct = default)
{
    // await using disposes the reader asynchronously
    // when the loop ends,
    // breaks early, throws, OR is cancelled – all
    // paths run DisposeAsync.
    await using var reader = new StreamReader(path);
    // ReadLineAsync(CancellationToken) (added in .NET
    // 7) returns ValueTask<string?>;
    // the `is { } line` pattern matches a non-null
    // line and ends the loop at EOF (null).
    while (await reader.ReadLineAsync(ct) is { } line)
        yield return line;
}

```

If a consumer writes `await foreach (var line in ReadLinesAsync(path)) { if (Found(line)) break; }`, the `StreamReader` is still disposed at the `break` because the compiler-generated `finally` in the `await foreach` invokes the enumerator's `DisposeAsync`, which runs the iterator's own `await using` cleanup. Forgetting `await` on `using` (writing a synchronous `using` over an `IAsyncDisposable`-only resource) is a compile error since the resource lacks a synchronous `Dispose` — the language nudges you toward the async form.

LINQ over async streams (.NET 10)

Before .NET 10, querying an `IAsyncEnumerable<T>` with `Where / Select / FirstOrDefaultAsync` required the community `System.Linq.Async` NuGet package. **.NET 10 ships `System.Linq.AsyncEnumerable` as part of the BCL**, providing the full standard operator set directly on `IAsyncEnumerable<T>`. You no longer reference the NuGet package when targeting .NET 10; the operators light up automatically from `System.Linq`.

```

using System.Linq; // AsyncEnumerable operators are
                  // in scope on .NET 10

// Stream, filter, project, and take – all lazily, all
// async, nothing materialized.
IAsyncEnumerable<string> hot =
    client.GetReposAsync("dotnet", token)
        .Where(r => r.StargazersCount > 1_000)
        .Select(r => r.Name)
        .Take(20);

await foreach (var name in
    hot.WithCancellation(token).ConfigureAwait(false))
    Console.WriteLine(name);

// Terminal operators are awaitable and end in *Async
// with a CancellationToken overload.
int popular = await client.GetReposAsync("dotnet",
    token)
    .CountAsync(r => r.StargazersCount > 1_000,
    token);

```

One migration wrinkle from the old NuGet library: the `*Await` and `*AwaitWithCancellation` operator families (e.g. `SelectAwait`) were renamed away. On .NET 10 you simply overload `Select` with an async lambda — `e.Select(async (int i, CancellationToken ct) => i * 2)` — instead of `e.SelectAwait(...)`. If you mechanically port code that referenced `System.Linq.Async`, expect a handful of method-name compile errors at exactly those call sites. The win is real: EF Core already exposes query results as `IAsyncEnumerable<T>`, so you can keep results in streaming form and apply LINQ without an intermediate `ToListAsync()`, lowering peak allocation in query-heavy services.

A word of caution on the laziness: operators like `OrderBy`, `Reverse`, and `ToListAsync` are *fully buffering* — they must drain the entire source before producing output. Putting `OrderBy` in the middle of a stream you intended to keep memory-bounded silently defeats the purpose. Streaming LINQ is only streaming for the streaming operators (`Where`, `Select`, `Take`, `Skip`, `TakeWhile`, `SelectMany`, ...).

`System.Threading.Channels` : push-based pipelines

Async streams are pull-based: one consumer drives one producer. When you need *fan-out* (many consumers), *fan-in* (many producers), or to decouple producer and consumer rates entirely, reach for `System.Threading.Channels`. A `Channel<T>` is an async, thread-safe, allocation-conscious queue split into a `ChannelWriter<T>` and a `ChannelReader<T>`.

The factory `Channel.CreateBounded<T>` / `Channel.CreateUnbounded<T>` returns the channel. The options object is where the performance decisions live:

- `SingleReader` / `SingleWriter` : assert that exactly one reader/writer touches the channel. These are *optimization promises*, not enforcement. Setting `SingleReader = true` lets the channel skip reader-side synchronization and pick a faster internal queue; violate it and you get races, not exceptions. Set them as tightly as your topology truly allows.
- `AllowSynchronousContinuations` : when `true`, a writer can run the reader's continuation inline on its own thread. Faster, but it lets producer and consumer code interleave on one thread and can cause unexpected re-entrancy. Default `false` is the safe choice.
- For bounded channels, `FullMode` selects the backpressure policy: `Wait` (the writer's `WriteAsync` suspends until space frees up — true backpressure), `DropOldest`, `DropNewest`, or `DropWrite`.

```

using System.Threading.Channels;

var channel = Channel.CreateBounded<WorkItem>(new
    BoundedChannelOptions(capacity: 100)
{
    FullMode = BoundedChannelFullMode.Wait,    //
        backpressure: block the producer
    SingleWriter = true,                        // one
        producer
    SingleReader = false,                       //
        multiple consumers (fan-out)
});

ChannelWriter<WorkItem> writer = channel.Writer;
ChannelReader<WorkItem> reader = channel.Reader;

```

The reader side exposes `ReadAllAsync(CancellationToken)`, which returns an `IAsyncEnumerable<T>` — so the consumer is just an `await foreach`, and channels compose with everything in the async-streams half of this chapter.

When you need finer control than `ReadAllAsync` gives — for example to read in batches, or to react to completion without an exception — drop to the lower-level pattern of `WaitToReadAsync` + `TryRead`. `WaitToReadAsync` returns `false` (rather than throwing) when the channel is completed and empty, which makes the drain condition explicit:

```

// Batched consumption: pull up to 32 items per turn
// to amortize per-item overhead.
var buffer = new List<WorkItem>(32);
while (await reader.WaitToReadAsync(ct))    // false
    => completed and drained
{
    while (buffer.Count < 32 && reader.TryRead(out var
        item)) // non-blocking drain
        buffer.Add(item);

    await FlushBatchAsync(buffer, ct);
    buffer.Clear();
}

```

`TryRead` never blocks: it returns `false` immediately when the buffer is momentarily empty, so the inner loop assembles whatever is available *right now* without waiting for a full batch. This

is the standard idiom for throughput-oriented consumers (bulk database inserts, network flushes) where per-item async overhead would dominate. Mirror it on the write side with `WaitToWriteAsync` + `TryWrite` when a producer wants to enqueue opportunistically without suspending.

Bounded producer/consumer with backpressure and multiple consumers

This worked example ties the channel concepts together: one producer feeding a bounded channel, N consumers draining it concurrently, real backpressure when consumers fall behind, and a clean completion handshake.

```

using System.Threading.Channels;

public sealed record WorkItem(int Id);

public static class Pipeline
{
    public static async Task RunAsync(int itemCount,
        int consumerCount, CancellationToken ct)
    {
        var channel = Channel.CreateBounded<WorkItem>
            (new BoundedChannelOptions(capacity: 16)
            {
                FullMode = BoundedChannelFullMode.Wait,
                // producer blocks when full
                SingleWriter = true,
                SingleReader = false,
            });

        // PRODUCER: a single task that writes items,
        // then completes the writer.
        Task producer = Task.Run(async () =>
        {
            try
            {
                for (int i = 0; i < itemCount; i++)
                {
                    // WriteAsync suspends here once
                    // 16 items are unread – backpressure.
                    // No memory growth: the producer
                    // literally cannot get ahead.
                    await
                    channel.Writer.WriteAsync(new WorkItem(i),
                    ct);
                }
            }
            finally
            {
                // Signal "no more items". Readers'
                ReadAllAsync loops will end.
                channel.Writer.Complete();
            }
        }, ct);

        // CONSUMERS: N tasks each draining
        ReadAllAsync. The channel hands each
        // item to exactly one consumer (work-stealing
        // fan-out), so no item is
        // processed twice and load balances across
        // consumers automatically.
        var consumers = new Task[consumerCount];
        for (int c = 0; c < consumerCount; c++)
    }
}

```

```

    {
        int consumerId = c;
        consumers[c] = Task.Run(async () =>
        {
            await foreach (var item in
channel.Reader.ReadAllAsync(ct))
            {
                await
Task.Delay(Random.Shared.Next(5, 25), ct);
                // simulate work
                Console.WriteLine($"consumer
{consumerId} -> item {item.Id}");
            }
        }, ct);
    }

    // Wait for the producer to finish writing AND
    // every consumer to drain.
    await producer;
    await Task.WhenAll(consumers);
}
}

```

Why this exhibits real backpressure: with `capacity: 16` and `FullMode.Wait`, the moment 16 unread items pile up, `WriteAsync` returns an incomplete `ValueTask` and the producer suspends. It resumes only when a consumer reads. The fast producer is throttled to consumer speed, and resident memory is bounded by `capacity` plus whatever each consumer holds in flight — *regardless of how many items flow through total*.

Pitfall — unbounded channels under a fast producer.

`Channel.CreateUnbounded<T>` never blocks `WriteAsync` (it returns a synchronously-completed `ValueTask`), which looks attractive for throughput. But if the producer outpaces the consumers — a JSON-deserializing producer feeding disk-writing consumers, say — the channel's internal queue grows without limit. There is no backpressure valve, so the only thing standing between you and an `OutOfMemoryException` is luck. Use unbounded channels only when you have an independent guarantee that the producer cannot durably outrun the consumer (e.g. a finite, small input), and prefer a bounded channel with `FullMode.Wait` as the default. The cost of bounding is one

`ValueTask` suspension; the cost of not bounding can be the process.

CancellationTokenSource : linked tokens, timeouts, CancelAsync

`CancellationTokenSource` (CTS) is the producer of tokens. Three composition tools matter for pipelines:

- `CancelAfter(TimeSpan)` arms a timer that cancels the source automatically. A CTS constructed with a `TimeSpan` is the same thing.
- `CreateLinkedTokenSource(params CancellationToken[])` returns a CTS that is cancelled when *any* of its inputs is cancelled (or when you cancel it directly). This is how you combine an ambient request-abort token with a per-operation timeout.
- `CancelAsync()` (introduced in .NET 8) cancels the source and returns a `Task` that completes only after all registered callbacks have finished running. Prefer it over `Cancel()` when you are *awaiting* an orderly shutdown, because `Cancel()` runs callbacks synchronously on the caller's thread and can deadlock if a callback re-enters your code or blocks.


```

// Combine the caller's abort token with a hard 30-
// second timeout.
public async Task<Repo[]>
    FetchWithDeadlineAsync(string org,
        CancellationToken requestAborted)
{
    using var timeout = new
        CancellationTokenSource(TimeSpan.FromSeconds(30));
    using var linked =
        CancellationTokenSource.CreateLinkedTokenSource(
            requestAborted, timeout.Token);

    try
    {
        return await client.GetReposAsync(org,
            linked.Token).ToArrayAsync(linked.Token);
    }
    catch (OperationCanceledException) when
        (timeout.IsCancellationRequested
            &&
            !requestAborted.IsCancellationRequested)
    {
        // Distinguish *why* we cancelled: timeout,
        // not a client disconnect.
        throw new TimeoutException($"Fetching repos
            for '{org}' exceeded 30s.");
    }
}

```

Note the disposal discipline: both the timeout CTS and the linked CTS are `using`-scoped. Failing to dispose a linked source leaks the registration that ties it to its parents — a classic slow leak in long-lived services where a parent token (e.g. an application-lifetime token) outlives thousands of short operations.

`Dispose` unregisters the link.

The `when` filter is the idiomatic way to recover the *cause* of a cancellation. A bare `catch (OperationCanceledException)` cannot tell a timeout apart from a client disconnect; inspecting `IsCancellationRequested` on each source disambiguates. `OperationCanceledException.CancellationToken` (present since the unified cancellation model in .NET Framework 4.0) also lets you compare `ex.CancellationToken == linked.Token`, but on a linked source that only tells you the *combined* token fired, not which input — and it is reliable only

when the throwing call actually populated the token (e.g. via `ThrowIfCancellationRequested`), which not every API does. Checking each source's `IsCancellationRequested` is the robust disambiguator.

Honoring tokens in library code

A library API that accepts a `CancellationToken` makes a promise: it will stop reasonably promptly when the token fires. Honoring it means weaving the token through every blocking or awaiting call (`HttpClient`, EF Core, `Stream`, `Task.Delay`), and inserting `ThrowIfCancellationRequested()` in tight CPU loops that have no natural await point. The cooperative model has no way to *forcibly* interrupt CPU-bound code — only cooperative checks work.

```
public async Task<int>
    CountMatchesAsync(IAsyncEnumerable<string>
        lines, string needle,
        CancellationToken ct)
{
    int count = 0;
    await foreach (var line in
        lines.WithCancellation(ct).ConfigureAwait(false))
    {
        ct.ThrowIfCancellationRequested(); // guard the CPU-bound loop body
        if (line.Contains(needle,
            StringComparison.OrdinalIgnoreCase))
            count++;
    }
    return count;
}
```

Pitfall — swallowing `OperationCanceledException`. The temptation, especially in a `finally`-heavy method, is to wrap everything in `try { ... } catch (OperationCanceledException) { }` and move on. This is almost always a bug. Cancellation is a *signal*, and swallowing it tells the caller “I completed normally” when in fact you were aborted. A caller that awaited your method now believes the

work finished, retries nothing, logs nothing, and may commit a transaction that should have rolled back. The correct default is to *let `OperationCanceledException` propagate*. Catch it only at a boundary where you are genuinely converting cancellation into a different, intentional outcome (an HTTP 499, a “user cancelled” UI state, a `TimeoutException` as above) — and never with an empty body. If you must observe-and-rethrow for logging, log at debug level and `throw;` .

A related subtlety: `Task.WhenAll` on cancelled tasks surfaces *one `OperationCanceledException`* , hiding sibling exceptions. When fanning out work that can both fail and cancel, prefer awaiting tasks individually or inspect `task.Exception` so a real fault isn’t masked by a cancellation.

Graceful shutdown: draining the channel and completing the writer

The completion handshake is the part people get wrong. A channel has two distinct end states, and conflating them loses data:

1. `Writer.Complete()` (or `Writer.TryComplete()`) means “no more items will ever be written.” It does *not* discard buffered items. After completion, `ReadAllAsync` keeps yielding everything still in the buffer, *then* ends the loop. This is graceful drain.
2. **Cancelling the consumers’ token** means “stop reading *now*”, abandoning buffered items. This is an abort, not a drain.

For a graceful shutdown you complete the writer and let readers run to natural exhaustion — you do **not** cancel the reader token. Here is the full pattern, including a cancellation-initiated shutdown that still drains in-flight work:

```

using System.Threading.Channels;

public sealed class IngestService(int workerCount)
{
    private readonly Channel<WorkItem> _channel =
        Channel.CreateBounded<WorkItem>(new
            BoundedChannelOptions(64)
            {
                FullMode = BoundedChannelFullMode.Wait,
                SingleWriter = false,    // multiple
                // producers may enqueue
                SingleReader = false,
            });

    private Task[] _workers = [];

    public void Start(Cancellation_token stopping)
    {
        _workers = Enumerable.Range(0, workerCount)
            .Select(_ => Task.Run(() =>
                ConsumeAsync(stopping)))
            .ToArray();
    }

    // Producers call this. Returns false if the
    // writer is already completed.
    public async ValueTask<bool>
        TryEnqueueAsync(WorkItem item,
            Cancellation_token ct)
    {
        // WaitToWriteAsync returns false once
        // Complete() has been called, true when
        // space is (or becomes) available. With
        // multiple writers (SingleWriter=false),
        // a sibling can grab that slot between the
        // wait and the TryWrite, so loop until
        // TryWrite succeeds or the writer completes –
        // don't assume a single attempt wins.
        while (await
            _channel.Writer.WaitToWriteAsync(ct).Configure
                Await(false))
        {
            if (_channel.Writer.TryWrite(item))    //
                non-blocking enqueue
                return true;
        }
        return false;    // writer was completed
    }

    private async Task ConsumeAsync(Cancellation_token
        ct)
    {

```

```

// IMPORTANT: do NOT pass `ct` to ReadAllAsync
// here. We want a *drain*:
// when the writer completes, this loop
// empties the buffer and ends.
// (Pass ct only if you want an abrupt stop
// that abandons buffered items.)
await foreach (var item in
_channel.Reader.ReadAllAsync())
{
    await ProcessAsync(item,
    ct).ConfigureAwait(false);
}

// Graceful shutdown: stop accepting new work,
// then let workers drain.
public async Task StopAsync()
{
    _channel.Writer.Complete();           // no
    more writes; buffer still drains
    await Task.WhenAll(_workers)         //
    wait for every buffered item to finish
    .ConfigureAwait(false);
}

private static ValueTask ProcessAsync(WorkItem
    item, CancellationToken ct) =>
    ValueTask.CompletedTask;           //
    real work goes here
}

```

The decision to *not* pass the cancellation token into `ReadAllAsync` is deliberate and is the crux of graceful shutdown. If you thread `ct` into the read loop and then cancel `ct`, the `await foreach` throws `OperationCanceledException` mid-buffer and you drop every item still queued. By completing the *writer* and awaiting the workers, every item that was accepted before shutdown is processed to completion. Reserve token-cancellation of the read loop for the *abort* path — a forced shutdown where losing in-flight work is acceptable (process is being killed anyway, SIGKILL is imminent). A robust service usually wants both: a soft stop (`Complete` + drain with a drain-deadline) and a hard stop (cancel the token) if the drain exceeds the deadline. The cleanest way to bound the drain is `Task.WaitAsync(TimeSpan)` (.NET 6+), which races the work-

ers against a timer and throws `TimeoutException` if they don't finish — without leaving a dangling `Task.Delay` behind:

```
public async Task StopAsync(TimeSpan drainTimeout)
{
    _channel.Writer.Complete();
    var drain = Task.WhenAll(_workers);

    // Race the drain against a timeout. Using
    // WaitAsync keeps it tidy: it cancels its
    // own internal timer when `drain` wins, so there
    // is no leftover pending Task.Delay.
    try
    {
        await
            drain.WaitAsync(drainTimeout).ConfigureAwait(false); // .NET 6+
    }
    catch (TimeoutException)
    {
        throw new TimeoutException("Workers did not
            drain within the deadline.");
    }
    // `drain` completed in time; awaiting it above
    // already surfaced any worker faults.
}
```

This separates the two concerns: `Complete()` guarantees termination of the read loops (they end when the buffer empties), and the timeout only governs *how long you are willing to wait* for that natural drain, never silently discarding work.

Channels vs. async iterators: choosing the right tool

These two abstractions overlap enough to confuse, but the decision is usually clear:

- Reach for an **async iterator** when there is one logical producer and one consumer, the producer is naturally lazy (paging, reading a file, an EF Core query), and structural back-pressure (consumer drives producer) is what you want. It allocates almost nothing and composes with LINQ.

- Reach for a **channel** when you need fan-out or fan-in, when producer and consumer must run *concurrently* at different rates, or when the producer is event-driven (a socket call-back, a timer) and has no natural “pull” point. Channels decouple the two sides with an explicit, tunable buffer.

A common hybrid: an event-driven producer writes to a bounded channel, and downstream code consumes

`channel.Reader.ReadAllAsync()` as an

`IEnumerable<T>`, then layers `.Where` / `.Select` (now BCL LINQ on .NET 10) on top. You get push-to-pull adaptation with backpressure and a clean streaming-query consumer — the best of both halves of this chapter.

Capstone: a cancellable, backpressured, LINQ-shaped pipeline

The following example combines every idea in the chapter: an async iterator producer (paged source,

`[EnumeratorCancellation]`), BCL LINQ shaping (.NET 10), a bounded channel as the fan-out buffer, multiple consumers, a linked CTS that fuses a caller token with a deadline, and a graceful drain on the success path versus an abort on timeout.

```

using System.Linq; // .NET 10 BCL
LINQ over IEnumerable<T>
using System.Threading.Channels;

public static class IngestPipeline
{
    public static async Task<int> RunAsync(
        GitHubClient client, string org, int
        consumers, CancellationToken caller)
    {
        // Fuse the caller's token with a 2-minute
        // hard deadline for the whole run.
        using var deadline = new
            CancellationTokenSource(TimeSpan.FromMinutes(2
            ));
        using var cts =
            CancellationTokenSource.CreateLinkedTokenSource
            (caller, deadline.Token);
        var ct = cts.Token;

        var channel = Channel.CreateBounded<Repo>(new
            BoundedChannelOptions(128)
            {
                FullMode = BoundedChannelFullMode.Wait,
                // backpressure to the producer
                SingleWriter = true,
                SingleReader = false,
            });

        // PRODUCER: stream + filter lazily, write
        // into the bounded channel.
        var producer = Task.Run(async () =>
        {
            try
            {
                IEnumerable<Repo> source =
                    client.GetReposAsync(org, ct)
                        .Where(r => r.StargazersCount >
                        0); // streaming operator
                await foreach (var repo in
                    source.WithCancellation(ct).ConfigureAwait(fal
                    se))
                {
                    await
                        channel.Writer.WriteAsync(repo,
                        ct).ConfigureAwait(false);
                }
            }
            finally
            {
                channel.Writer.Complete(); // always
                // signal completion, even on throw
            }
        }, ct);
    }
}

```



```

// CONSUMERS: drain WITHOUT a token so
// Complete() triggers a graceful drain.
int processed = 0;
var workers = Enumerable.Range(0,
consumers).Select(_ => Task.Run(async () =>
{
    await foreach (var repo in
channel.Reader.ReadAllAsync()).ConfigureAwait(f
alse))
        Interlocked.Increment(ref processed);
    // real work would go here
})).ToArray();

try
{
    await producer.ConfigureAwait(false);
    // surface producer faults
    await
Task.WhenAll(workers).ConfigureAwait(false);
    // let buffer drain fully
    return processed;
}
catch (OperationCanceledException) when
(deadline.IsCancellationRequested
&&
!caller.IsCancellationRequested)
{
    // Only a deadline-induced cancel becomes
    // TimeoutException. If the *caller*
    // cancelled (even if the deadline also
    // fired in a race), this filter is false
    // and the OperationCanceledException
    // propagates untouched – caller intent wins.
    throw new TimeoutException($"Ingest of
'{org}' exceeded the 2-minute deadline.");
}
}
}

```

The load-bearing decisions: the producer threads `ct` through both the iterator (which forwards it via `[EnumeratorCancellation]`) and `WriteAsync`, so cancellation propagates end to end; `Complete()` lives in a `finally` so a producer fault still releases the consumers; the consumers read *without* a token so they finish by draining the buffer rather than throwing mid-stream; and the `when` filter converts a deadline-induced cancel into a typed `TimeoutException` while letting a caller-initiated cancel propagate as

`OperationCanceledException` untouched. Swap `consumers` from 1 to 8 and the only thing that changes is throughput — correctness and the memory ceiling (128 buffered repos) are invariant.

Performance notes

- **State-machine allocation.** An async iterator boxes its state machine once when an `await` first suspends. Streams that complete synchronously (everything cached) stay allocation-free until the first real suspension.
`ValueTask` / `ValueTask<bool>` returns from `MoveNextAsync` avoid per-element `Task` allocations — this is why hot async streams don't generate a `Task` per element.
- **Channel internals.** Bounded channels use an array-backed ring buffer; unbounded use a segmented linked queue. Both pool their await machinery (`IValueTaskSource`) so a steady-state read/write does not allocate per operation. The `SingleReader` / `SingleWriter` flags switch to lock-free or lighter-locking implementations — set them honestly for a measurable win under contention.
- `AllowSynchronousContinuations = true` removes a thread hop on the read completion but lets the writer's thread run consumer code; only enable it when you have profiled the hop as a bottleneck and the consumer continuation is short and non-reentrant.
- `ConfigureAwait(false)` on `await foreach` saves a context capture/post per element on context-bound runtimes; on context-free runtimes it is free either way, so the cost of always writing it in library code is zero.
- **Cancellation registration** is not free: each `CreateLinkedTokenSource` and each downstream call that registers a callback adds a linked-list node. In a hot loop, prefer checking `ThrowIfCancellationRequested()` (a single volatile read) over creating fresh linked sources per iteration.

20 · Threading, the new Lock Type, Interlocked, and Parallelism

This chapter is about the *bottom* of the concurrency stack — the primitives that `async / await`, channels, and dataflow are ultimately built on. We will spend most of our time on the `System.Threading.Lock` type that C# 13 added and the language pattern that makes it idiomatic, then descend into lock-free `Interlocked` work and the .NET memory model, and finally come back up to coarse-grained data parallelism with `Parallel` and PLINQ. The recurring theme is *choosing the cheapest correct primitive*: a `Lock` is cheaper than a `SemaphoreSlim`, an `Interlocked.Increment` is cheaper than a `Lock`, and frequently the cheapest correct primitive is no shared mutable state at all.

The problem `lock(object)` always had

For two decades the canonical mutual-exclusion idiom was a plain `object` and the `lock` statement:

```
private readonly object _sync = new();

public void Add(int x)
{
    lock (_sync)           // lowered to Monitor.Enter /
        Monitor.Exit
    {
        _total += x;
    }
}
```

The compiler lowered that `lock` block to roughly:

```

object lockObj = _sync;
bool lockTaken = false;
try
{
    Monitor.Enter(lockObj, ref lockTaken);
    _total += x;
}
finally
{
    if (lockTaken) Monitor.Exit(lockObj);
}

```

`Monitor` is a per-object facility implemented on the object header (the *sync block index*). That design has real costs and real footguns. Every reference-type object in the CLR reserves header bits that can hold a hash code, GC state, or — when used as a lock — an index into the runtime’s sync-block table. The first time an object is contended, the runtime *inflates* it: it allocates a full sync block, migrates the thin-lock state into it, and from then on every enter/exit pays an indirection through that table. So the cost of `lock(object)` is not fixed; it depends on the history of contention on the specific object instance. Worse, that instance is a regular heap object the GC tracks, and locking on it commingles your synchronization concern with the object’s identity.

The footguns compound from there. Because *any* reference could be passed to `Monitor.Enter`, people locked on `this` (exposing the lock to every caller that holds a reference to the instance), on `typeof(Foo)` (a process-wide `Type` object shared across unrelated code), on interned string literals (`lock("sync")` deadlocks against any other code that locked the same interned literal anywhere in the process), and on boxed value types (a fresh box each time, so the lock protects nothing). Each is a latent deadlock or silent-no-op, and none of them is a syntax error. There is also no type-level signal that a field is “the lock”; a reviewer staring at `private readonly object _x` cannot tell a synchronization object from a cache key or a sentinel. The `Lock` type fixes every one of these at the type level: it is purpose-built, it is its own kind, and the language refuses to let it masquerade as a general `object`.

System.Threading.Lock (C# 13 / .NET 9)

.NET 9 shipped a dedicated reference type, `System.Threading.Lock`, and C# 13 taught the compiler to recognize it. The type is small and intention-revealing:

```
using System.Threading;

public sealed class Ledger
{
    private readonly Lock _gate = new();    // C# 13 /
                                           // .NET 9: a real lock type
    private long _total;

    public void Add(int x)
    {
        using (_gate.EnterScope())          // enters;
        {                                     Dispose() exits
            {
                _total += x;
            }
        }

        public long Total
        {
            get
            {
                using (_gate.EnterScope())
                {
                    return _total;
                }
            }
        }
    }
}
```

`EnterScope()` enters the lock and returns a `Lock.Scope` value whose `Dispose()` exits it. `Lock.Scope` is a `ref struct` — that is deliberate. Because a `ref struct` cannot be boxed, cannot be stored in a field, cannot escape to the heap, and cannot cross an `await` or `yield`, the compiler *statically* prevents you from holding the scope across an asynchronous suspension or leaking it out of the method. The lock release is tied to a stack-scoped lifetime that the JIT and the language both understand.

The `Lock` API is richer than `lock` / `Monitor` exposed conveniently:

```
var gate = new Lock();

if (gate.TryEnter()) // non-blocking
    attempt
{
    try { /* ... */ }
    finally { gate.Exit(); }
}

if (gate.TryEnter(TimeSpan.FromMilliseconds(50))) //
    bounded wait
{
    try { /* ... */ }
    finally { gate.Exit(); }
}

bool held = gate.IsHeldByCurrentThread; //
    useful in asserts
```

`Lock` is reentrant in exactly the way `Monitor` is: a thread that already holds it can enter again, and the lock is only released when the matching number of exits have run. `EnterScope` increments and `Scope.Dispose` decrements that recursion count, and `Exit` throws `SynchronizationLockException` if the calling thread does not hold the lock — a useful tripwire that catches mismatched enter/exit pairs that the old `Monitor.Exit` would silently corrupt. Do not rely on reentrancy as a design tool. Recursive locking usually means an invariant is being re-checked in a path that already holds the lock, and it hides a real risk: a method that *publishes* a partially-updated object while still holding the lock can be re-entered by a callback that observes the broken invariant, because reentrancy grants access without the data being consistent. Treat reentrancy as a compatibility behavior, not a feature — but because it matches the old `Monitor` semantics exactly, migrating `object` to `Lock` never changes run-time behavior, only performance and clarity.

It is worth being precise about what the `ref struct` buys you beyond preventing `await`-while-holding. A `ref struct` is stack-only by the language rules introduced and refined across

C# 11–13: it cannot be captured by a lambda, cannot be boxed, and cannot be a field of a non- `ref` struct or class. It cannot be kept *alive across* an `await` or a `yield return` either — and note the precise rule, because C# 13 relaxed it: you may now *declare* a `ref struct` local (or a `ref` local) inside an `async` or iterator method, you simply cannot let it survive a suspension point, since the compiler cannot hoist it into the state machine. That distinction is exactly why holding a `Lock.Scope` across an `await` is a compile error while a `Lock.Scope` confined to a synchronous span *within* an `async` method is perfectly legal. Every one of these restrictions corresponds to a real way people leaked monitor locks in the old world — a `Monitor.Enter` whose matching `Exit` lived in a captured closure, or in a continuation, or in a finalizer. With `Lock.Scope` the compiler proves at build time that the release happens on the same stack frame, on the same thread, before the frame unwinds. That is a genuine correctness upgrade encoded in the type system, not just a convenience.

The `lock` pattern: `lock (aLock)` over a `Lock`

You do not have to write `using (_gate.EnterScope())`. C# 13 added a **lock pattern**: when the operand of a `lock` statement is statically typed as `System.Threading.Lock`, the compiler does *not* lower to `Monitor.Enter`. Instead it lowers to `EnterScope`:

```
private readonly Lock _gate = new();

public void Add(int x)
{
    lock (_gate)           // C# 13 lock-pattern: lowered
                           to EnterScope, NOT Monitor
    {
        _total += x;
    }
}
```

is lowered to the equivalent of:

```

Lock.Scope scope = _gate.EnterScope();
try
{
    _total += x;
}
finally
{
    scope.Dispose();
}

```

Contrast that with the `Monitor`-based lowering shown earlier for `lock (object)`. The `Lock` path has no `bool lockTaken`, no `object` argument, and no sync-block indirection — it calls a method on a purpose-built type that maintains its own thin, *fu-tex*-style wait state. This is the single most important migration in the chapter: **change the field type from `object` to `Lock` and the existing `lock` statements get faster and safer with zero call-site edits**. The IL changes under you; the C# does not.

There is a sharp edge here that the compiler will warn you about. `System.Threading.Lock` is a `sealed class` — a *reference* type — so converting it to `object` does **not** box it; it produces another reference to the very same instance. The hazard is not a stale copy but a stale *mechanism*: if you pass that reference where an `object` is expected — `Monitor.Enter(_gate)`, `lock ((object)_gate)`, or storing it in an `object` local and locking that — the `lock / Monitor` path uses the object-header sync-block monitor *on the `Lock` instance* instead of the `Lock`'s own `EnterScope` state machine. Those are two different, non-interoperating locks living on the same object: a thread inside `_gate.EnterScope()` and a thread inside `Monitor.Enter((object)_gate)` do **not** exclude each other. C# 13 emits **CS9216** (“a value of type `System.Threading.Lock` converted to a different type will use likely unintended monitor-based locking in `lock` statement”) whenever a `Lock` is implicitly (or explicitly) converted to `object`, `dynamic`, or an interface it implements — not only inside a `lock` statement. Treat that warning as an error.


```
private readonly Lock _gate = new();

object o = _gate;           // CS9216 – same instance,
                             // but now seen as plain object
lock (o) { }                 // Monitor on the Lock's
                             // object header – a DIFFERENT lock from
                             // _gate.EnterScope()
```

Pitfall: bringing the `lock(object)` mindset to a `Lock`

Two habits from the `object` era are now actively wrong:

- **Exposing the lock.** With `object` locks people sometimes returned the sync object or made it `protected` so subclasses could “coordinate.” Doing that with a `Lock` (or an `object`) lets external code take your lock and starve or deadlock you; the lock is an implementation detail and must stay `private`. Keep it `private readonly Lock`, never a property, never a parameter.
- **Upcasting it to reuse old helpers.** Any code path that converts the `Lock` to `object` — passing it to a method typed `object`, putting it in a `List<object>`, using it as a dictionary key, then locking that reference — reverts to monitor semantics on the `Lock` instance’s object header, a lock unrelated to its `EnterScope` state. (`Lock` is a class, so this is a reference upcast, not a box — but the synchronization is just as broken.) The CS9216 warning catches the obvious cases; the subtle ones (generic methods constrained to `class`, `params object[]`) you must catch in review.

When *not* to use `Lock`

`Lock` is synchronous and thread-affine. It blocks the calling thread, and you cannot hold it across `await` (the `ref struct` scope makes that a compile error, which is the design working as intended). If your critical section is asynchronous — it awaits I/O while holding exclusivity — you need `SemaphoreSlim(1, 1)` with `await WaitAsync() / Release()`, or a higher-level construct like a `Channel<T>` that serializes by construction. Reaching for `Lock` around an `await` is a category error; reaching for `SemaphoreSlim` around a purely-CPU critical section is

needless overhead. Despite its name, `SemaphoreSlim`'s uncontended path stays in user mode (it spins briefly before parking), but a contended `WaitAsync` allocates and schedules a `Task` continuation, and its synchronous `Wait` parks on an internally managed wait handle. That async/`Task` machinery — plus the lack of an owning thread and of reentrancy — is pure overhead when the critical section never awaits; a `Lock` is a thinner, thread-affine primitive for that case.

There is a deeper reason the two are not interchangeable beyond “one is async.” `Lock` is *thread-affine and reentrant*: it tracks which thread holds it, so it can detect reentrancy and refuse a mismatched exit. A `SemaphoreSlim` has no concept of an owning thread — it is a counter — so it is *not* reentrant (a second `WaitAsync` from the same logical flow that already holds it will deadlock), and `Release()` can be called from any thread, including one that never waited. That flexibility is exactly what async needs, because the continuation after `await` may resume on a *different* thread than the one that took the lock, and a thread-affine primitive cannot survive that thread hop. So the rule is not stylistic: a thread-affine lock is correct precisely when there is no thread hop (synchronous critical sections), and a thread-agnostic gate is required precisely when there is (async critical sections). The compile error you get from trying to hold a `Lock.Scope` across `await` is the type system encoding this invariant for you.

Interlocked: lock-free counters and flags

When the entire critical section is a single read-modify-write on one machine-word field, a lock is overkill. The `System.Threading.Interlocked` class exposes atomic operations that compile to a single CPU instruction (e.g. `lock xadd`, `lock cmpxchg` on x86-64) with no kernel transition and no contention object. Overloads exist for `int`, `long`, `uint`, and `ulong` (the unsigned ones since .NET 5), plus `Exchange / CompareExchange` for `object`, `IntPtr`, `float`, and `double`.

A correct lock-free counter:

```
public sealed class RequestCounter
{
    private long _count;

    public void Hit() => Interlocked.Increment(ref
        _count);    // atomic ++ , returns the NEW
                    // value

    // A long read is atomic on 64-bit but NOT on 32-
    // bit; Interlocked.Read reads it
    // atomically on every platform (and acts as a
    // full fence against stale caching).
    public long Count => Interlocked.Read(ref _count);
    // atomic read of a long
}
```

`Increment` / `Decrement` / `Add` cover accumulation. The real workhorse, though, is `CompareExchange` (CAS — compare-and-swap), which is how you build *any* lock-free update of a value computed from its previous state. The pattern is a read-compute-CAS retry loop:

```
public sealed class RunningMax
{
    private long _max = long.MinValue;

    public void Offer(long candidate)
    {
        long snapshot;
        do
        {
            snapshot = Volatile.Read(ref _max);
            if (candidate <= snapshot) return;
            // nothing to do
            // CompareExchange writes `candidate` ONLY
            // IF _max still equals snapshot.
            // It returns the value that was in _max
            // at the moment of the call.
        }
        while (Interlocked.CompareExchange(ref _max,
            candidate, snapshot) != snapshot);
        // Loop repeats only if another thread changed
        // _max between our read and our CAS.
    }
}
```

The loop is the essence of lock-free programming: you take a snapshot, compute a new value, and atomically commit it *only if nobody changed the field in the meantime*. If they did,

`CompareExchange` returns the new current value (not your snapshot), the `!=` is true, and you retry with fresh data. Under low contention this loop almost never spins; under high contention it can livelock-starve, which is the tradeoff against a lock's fairness.

One subtle hazard of CAS loops is the **ABA problem**: if the field changes from value A to B and back to A between your read and your CAS, the `CompareExchange` succeeds even though the world moved underneath you. For a monotonically increasing counter or a max this is harmless, but for a field that holds a *pointer to a node* (a lock-free stack head, say) ABA can corrupt the structure — a node you popped can be pushed back and your stale CAS will splice it incorrectly. The standard mitigations are a version-tagged pointer (pack a pointer and a monotonic counter into a double-width CAS, or into a single `long` if the pointer fits) or, more pragmatically, to stop hand-rolling lock-free data structures and use the ones in

`System.Collections.Concurrent`, which the BCL team has already gotten right. Lock-free programming is correct in the small and treacherous in the large; reach for it for counters and flags, and be very deliberate before building anything with linked structure.

For boolean flags, `Interlocked.Exchange` gives you “set and tell me the previous value” — the idiom for run-once gates:

```
public sealed class OneShot
{
    private int _done;    // 0 = not yet, 1 = done;
                        // Interlocked has no bool overloads

    public bool TryClaim() =>
        Interlocked.Exchange(ref _done, 1) == 0;    //
                        // first caller gets true, rest false
}
```

`Interlocked.Or` and `Interlocked.And` (since .NET 5) atomically merge bit flags without a CAS loop — useful for `[Flags]` enums backed by `int` / `long` :

```
private int _state;
// atomically set a bit, return previous value
int prev = Interlocked.Or(ref _state, 0b0010);
```

When not to use Interlocked

`Interlocked` is correct only when the invariant spans a **single** field. The moment you need two fields to change together (a head and a tail pointer, a value and its version), a sequence of `Interlocked` calls is *not* atomic as a unit and you have a torn-update bug. Either pack both into one `long` and CAS the pair, or take a `Lock` . Do not chain `Interlocked` operations and hope.

The memory model: `volatile` , `Volatile` , and barriers

Atomicity is not the same as visibility. The CPU and the JIT may reorder reads and writes, and a write by one core may sit in a store buffer invisible to another core for a while. The classic broken double-checked pattern:

```
private bool _initialized;
private Config? _config;

void Init()
{
    _config = LoadConfig();    // (A)
    _initialized = true;        // (B) - may become
                                visible BEFORE (A) without a barrier
}

Config? Get() => _initialized ? _config : null;    //
                                                    may see _initialized==true, _config==null
```

Without ordering guarantees a reader can observe `_initialized == true` while `_config` is still `null` , because the two writes can be reordered or become visible out of order. The fix is acquire/release semantics.

`Volatile.Read` / `Volatile.Write` (or a `volatile` field) impose half-fences: a `Volatile.Write` is a *release* — everything before it is visible before the write becomes visible — and a `Volatile.Read` is an *acquire* — nothing after it floats above the read.

```
void Init()
{
    _config = LoadConfig();
    Volatile.Write(ref _initialized, true);    //
                                              // release: publishes _config too
}

Config? Get() =>
    Volatile.Read(ref _initialized) ? _config : null;
    // acquire: sees the published _config
```

The .NET memory model (formalized for ECMA and tightened in modern .NET) guarantees: ordinary reference/ `int` / `long` - on-64-bit reads and writes are *atomic* but *not ordered*; `volatile` / `Volatile.*` give acquire/release; `Interlocked.*` operations are full fences; and `Thread.MemoryBarrier()` is a standalone full fence. Two caveats the formal model makes explicit and that people get wrong. First, atomicity is per-type-and-platform: a `long` or `double` read is *not* guaranteed atomic on 32-bit runtimes, which is exactly why `Interlocked.Read(ref long)` exists — on 64-bit it is essentially free, on 32-bit it does the work to read the two halves consistently. Second, `volatile` is a property of the *access site*, not of the value: marking a field `volatile` makes every read an acquire and every write a release, but it does nothing for a `ref` to that field passed elsewhere, and it cannot make a multi-word update atomic. It orders, it does not bundle.

In practice you almost never reach for raw barriers — prefer `Lock` (which fences on enter and exit), `Interlocked`, `Volatile`, or a `Lazy<T>` that handles publication for you. The runtime’s strong model on x86/x64 (where ordinary stores already have release-ish semantics) hides many missing barriers, which is precisely why memory-model bugs are so dangerous: code that “works” on your x64 dev box can tear on ARM64,

where the hardware reorders far more aggressively. If you target ARM (Apple Silicon, Graviton, mobile), assume nothing about ordering that you have not enforced with `Volatile`, `Interlocked`, or a lock. The `volatile` keyword is appropriate for a lone flag polled across threads; for anything structured, a higher-level primitive is clearer, portable, and just as fast.

Parallel.For / ForEach / ForEachAsync

When you have a large collection of independent CPU-bound work items, `System.Threading.Tasks.Parallel` partitions them across the thread pool for you. `Parallel.For` and `Parallel.ForEach` are synchronous and block until all partitions finish; they shine for embarrassingly parallel CPU work.

```
double[] input = GetSamples();
double[] output = new double[input.Length];

Parallel.For(0, input.Length, new ParallelOptions
{
    MaxDegreeOfParallelism =
        Environment.ProcessorCount
}, i =>
{
    output[i] = ExpensiveTransform(input[i]); //
    each i is independent – no shared writes
});
```

The cardinal rule: the body must be free of cross-iteration data races. Writing to `output[i]` is fine because each iteration owns a distinct index; accumulating into a shared variable is not. For reductions, use the thread-local overload so each worker accumulates privately and you merge once at the end:

```

long total = 0;
Parallel.For(0, input.Length,
    () => 0L, //
    (i, _, localSum) => localSum + (long)input[i], //
    accumulate locally
    localSum => Interlocked.Add(ref total, localSum));
    // merge once per thread
// One Interlocked.Add per worker thread, not per
    element – contention stays tiny.

```

For **I/O-bound** parallelism, `Parallel.ForEachAsync` (introduced in .NET 6) is the right tool: it runs an async delegate over a source with a bounded degree of parallelism and a cancellation token, and it does *not* burn a thread per in-flight operation the way a parallel loop full of `.Result` calls would.

```

public static async Task<IReadOnlyList<Page>>
    FetchAllAsync(
        IEnumerable<Uri> urls,
        HttpClient http,
        CancellationToken ct)
{
    var results = new ConcurrentBag<Page>();

    var options = new ParallelOptions
    {
        MaxDegreeOfParallelism = 8, // at most 8
        requests in flight at once
        CancellationToken = ct
    };

    await Parallel.ForEachAsync(urls, options, async
        (url, token) =>
    {
        // `token` is linked to ct AND cancelled if a
        sibling iteration faults.
        using HttpResponseMessage resp = await
            http.GetAsync(url, token);
        string body = await
            resp.Content.ReadAsStringAsync(token);
        results.Add(new Page(url, body));
    });

    return results.ToArray();
}

```


Two details matter. First, `MaxDegreeOfParallelism = 8` is a *concurrency* limit, not a thread count — because the body is *async*, those eight slots are filled by continuations, not blocked threads, so the cost of raising the limit is roughly proportional to outstanding I/O rather than to OS threads. Second, the inner `token` is a *linked* token: it triggers both when your `ct` fires and when any iteration throws, which cancels the in-flight siblings. If a body throws, `Parallel.ForEachAsync` collects the exceptions and surfaces an `AggregateException` (commonly unwrapped to the first via `await`) after the in-flight work drains. Use `ConcurrentBag` / `ConcurrentQueue` (or write to disjoint indices) for output — the body runs concurrently.

Partitioning and chunk granularity

The synchronous `Parallel.For` / `ForEach` loops do not hand one element at a time to each worker — that would drown in scheduling overhead for cheap bodies. They use a `Partitioner` to slice the source into *ranges* (for indexable sources) or *chunks* (for `IEnumerable`), and each worker pulls a chunk, processes it, and comes back for more. The default chunk partitioner grows chunk sizes geometrically, which balances two competing pressures: large chunks amortize the cost of the work-stealing handshake, while small chunks keep the workers from finishing at wildly different times (load imbalance) when per-element cost varies. You can take control when the default guesses wrong:

```
// Coarse, fixed-size ranges: great when each element
// is uniform and cheap.
var partitioner = Partitioner.Create(0, source.Length,
    rangeSize: 4096);
Parallel.ForEach(partitioner, range =>
{
    for (int i = range.Item1; i < range.Item2; i++)
        // (Item1, Item2) = [start, end)
        Process(source[i]);
});
```

Tuning the partitioner is the right lever when you see one of two symptoms in a profiler: workers idling at the end of a loop (chunks too big — work imbalance) or high scheduler overhead

relative to body time (chunks too small). For uniform, very cheap bodies, a coarse range partitioner like the one above often doubles throughput over the default by eliminating per-chunk coordination. For wildly variable bodies, the default geometric chunking usually wins because it re-balances near the end.

PLINQ: AsParallel and when it earns its keep

PLINQ parallelizes a LINQ query by adding `.AsParallel()`. It is genuinely useful for *CPU-bound, side-effect-free* projections and filters over large in-memory sequences:

```
double[] result = numbers
    .AsParallel()
    .Where(n => IsPrime(n))           // each predicate
    is pure and CPU-heavy
    .Select(n => Math.Sqrt(n))
    .ToArray();
```

By default PLINQ does **not** preserve order (it partitions and runs out of sequence). If you need input order in the output, opt in with `.AsOrdered()` — but that imposes buffering and re-assembly overhead, so only ask for ordering you actually need:

```
var ordered =
    numbers.AsParallel().AsOrdered().Where(IsPrime)
    .ToArray();
```

PLINQ helps when (1) the per-element work is substantial enough to dwarf partitioning/merging overhead, (2) the source is large, and (3) the operations are pure. It *hurts* when the work is cheap (the coordination overhead exceeds the savings), when the body does I/O (use `Parallel.ForEachAsync` instead), or when the body has side effects into shared state (you reintroduce all the synchronization you were trying to avoid). For small or cheap sequences plain LINQ is faster; measure before sprinkling `.AsParallel()`.

PLINQ also exposes knobs the synchronous loops hide.

`.WithDegreeOfParallelism(n)` caps worker count;

`.WithMergeOptions(ParallelMergeOptions.NotBuffered)`

streams results as they are produced instead of buffering the whole output, which matters when a downstream consumer can start early; and

`.WithExecutionMode(ParallelExecutionMode.ForceParallel)` overrides PLINQ's own heuristic, which by default may decide a query is too cheap and run it sequentially. That last point surprises people: simply writing `.AsParallel()` does *not* guarantee parallel execution — PLINQ analyzes the query shape and silently falls back to sequential when it predicts parallelism would lose, which is usually the right call but occasionally not the one you intended.

```
var result = numbers
    .AsParallel()
    .WithDegreeOfParallelism(4)

    .WithExecutionMode(ParallelExecutionMode.ForceParallel)
    .Where(IsPrime)
    .ToArray();
```

A final caution: an exception thrown inside any PLINQ delegate is wrapped in an `AggregateException` (one or more inner exceptions, since several partitions may fault), so PLINQ query consumers must `catch (AggregateException)` or use `.Flatten().Handle(...)`, not the bare exception type they would catch from sequential LINQ.

ConcurrentDictionary: patterns and the GetOrAdd trap

`ConcurrentDictionary<TKey, TValue>` is the default associative store for concurrent code. Its reads are lock-free and its writes are striped across a small set of internal locks (the *lock count*, tunable in a constructor overload and defaulting to a multiple of the processor count), so it scales far better than a `Dictionary` behind a single `Lock`. The striping is the whole point: two writers touching keys that hash to different stripes proceed in parallel, and readers never block writers at all be-

cause reads walk the bucket chain without taking any lock. That makes it an excellent choice for read-heavy, write-occasional caches — but it also means a `ConcurrentDictionary` carries more memory overhead per entry than a plain `Dictionary` (extra padding, the stripe locks, volatile node links), so it is the wrong default for a dictionary that is only ever touched by one thread. Most operations are correctly atomic: `TryAdd`, `TryRemove`, `TryUpdate`, and the `AddOrUpdate` / `GetOrAdd` family give you compound operations without an external lock — and crucially, each of those is atomic *as a single call*, but a *sequence* of them is not, which is the seam where most misuse happens.

But there is a notorious subtlety, and it is one of the most common concurrency bugs in real codebases.

Pitfall: `GetOrAdd`'s `valueFactory` can run more than once

`GetOrAdd(key, valueFactory)` guarantees that **exactly one value is stored** and returned to all callers — but it does **not** guarantee the `valueFactory` runs only once. If two threads call `GetOrAdd` for a missing key simultaneously, both may invoke the factory; the dictionary then keeps one result and *discards* the other. The stored value is consistent, but any side effect inside the factory (opening a connection, registering a handler, spinning up a worker) may have happened twice, and the discarded object may need disposal that never happens.

```
private readonly ConcurrentDictionary<string,
    HttpClient> _clients = new();

public HttpClient ForHost(string host) =>
    _clients.GetOrAdd(host, h =>
    {
        // DANGER: under a race this factory can run
        // twice for the same host.
        // One HttpClient is stored; the other is
        // silently abandoned (and leaked).
        Console.WriteLine($"Creating client for {h}");
        return new HttpClient { BaseAddress = new
            Uri($"https://{h}") };
    });
```

The fix is to make the factory cheap and idempotent, and put the *expensive, must-run-once* work behind a `Lazy<T>`. Store `Lazy<HttpClient>` in the dictionary: `GetOrAdd` may construct several `Lazy` wrappers, but a `Lazy` is trivially cheap, only one wrapper wins the slot, and `.Value` on that single winner forces the real factory **exactly once** (and `LazyThreadSafetyMode.ExecutionAndPublication`, the default, serializes that).

```
private readonly ConcurrentDictionary<string,
    Lazy<HttpClient>> _clients = new();

public HttpClient ForHost(string host)
{
    Lazy<HttpClient> lazy = _clients.GetOrAdd(host, h
=>
    new Lazy<HttpClient>(() =>
    {
        // Runs exactly once for the winning Lazy,
        // for this host.
        Console.WriteLine($"Creating client for
{h}");
        return new HttpClient { BaseAddress = new
Uri($"https://{h}") });
    }));

    return lazy.Value; // forces the real factory
                       // once; all callers share the result
}
```

Throwaway `Lazy` wrappers that lose the race are never `.Value`-forced, so their expensive factory never runs and there is nothing to leak. This `ConcurrentDictionary<TKey, Lazy<TValue>>` idiom is the standard fix and should be your default whenever the factory is expensive or has side effects.

A second `ConcurrentDictionary` pitfall: there is no atomic “iterate and mutate.” Enumerating the dictionary gives a moment-in-time snapshot that may not reflect concurrent writes, and `Count` is a stop-the-world operation that acquires all internal locks — avoid it on hot paths. For “increment a counter per key,” prefer `AddOrUpdate(key, 1, (_, v) => v + 1)` over a `TryGetValue` / `+1` / `TryUpdate` dance you write yourself. Note the same caveat as `GetOrAdd`: under contention `AddOrUpdate`

may invoke the update factory more than once (it spins on an internal CAS), so the *net effect* is correct only because the delegate is a pure function of the current value and gets re-read on retry — never put a side effect in that factory. The stored result is consistent; the delegate is not guaranteed single-shot.

A worked example: a bounded, observable scraper

The following ties the chapter together: bounded async parallelism (`Parallel.ForEachAsync`), a lock-free metric (`Interlocked`), exclusive critical-section work with the new `Lock` and the C# 13 lock-pattern, and a per-key one-time resource via `ConcurrentDictionary<..., Lazy<...>>` .

```

using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

public sealed class HostScraper
{
    private readonly Lock _logGate = new();
    // C# 13 Lock
    private readonly ConcurrentDictionary<string,
        Lazy<HttpClient>> _clients = new();
    private long _bytesFetched;
    // lock-free counter
    private int _failures;

    private HttpClient ClientFor(Uri uri) =>
        _clients.GetOrAdd(uri.Host, host =>
            new Lazy<HttpClient>(() =>
                // factory runs once
                new HttpClient { Timeout =
                    TimeSpan.FromSeconds(10) })).Value;

    public async Task RunAsync(IEnumerable<Uri> urls,
        CancellationToken ct)
    {
        var options = new ParallelOptions
        {
            MaxDegreeOfParallelism = 6,
            // at most 6 in flight
            CancellationToken = ct
        };

        await Parallel.ForEachAsync(urls, options,
            async (uri, token) =>
            {
                try
                {
                    byte[] body = await
                        ClientFor(uri).GetByteArrayAsync(uri, token);
                    Interlocked.Add(ref _bytesFetched,
                        body.Length); // atomic accumulate

                    Lock(_logGate)
                    // lock-pattern -> EnterScope
                    {
                        Console.WriteLine($"{uri} ->
                            {body.Length} bytes"); // serialize console
                        writes
                    }
                }
                catch (Exception ex) when (ex is not
                    OperationCanceledException)
                {

```

```

        Interlocked.Increment(ref _failures);
        // atomic flag/count
    }
    });
}

public (long Bytes, int Failures) Snapshot() =>
    (Interlocked.Read(ref _bytesFetched),
     Volatile.Read(ref _failures));
}

```

Every synchronization choice here is the cheapest correct one. The byte total and failure count are single-field read-modify-writes, so they are `Interlocked`, not a lock. The console is a shared, non-thread-safe resource where interleaved writes would garble output, so it gets a real critical section — and because `_logGate` is typed `Lock`, the `lock (_logGate)` statement lowers to `EnterScope`, not `Monitor`. The per-host `HttpClient` must be created once and shared, so it lives behind `ConcurrentDictionary<string, Lazy<HttpClient>>`. And the whole thing is bounded async parallelism, not a thread-per-URL fan-out, so 10,000 URLs cost six concurrent requests' worth of resources, cancel cleanly through the linked token, and never block a pool thread on I/O.

Performance notes and choosing the model

- **Lock vs Monitor**. The `Lock` fast path (uncontended acquire) is a thin interlocked operation on the type's own state with no object-header inflation and no sync-block table lookup. For tight, frequently-taken locks it measurably beats `lock(object)`, and it never accidentally inflates a shared object's header. Migrate object locks to `Lock`; the call sites do not change.
- **Interlocked vs Lock**. A single `Interlocked` op is a cache-line-bounded atomic with no kernel involvement and no fairness machinery; under low contention it is several times cheaper than acquiring a lock. Under *high* contention on one cache line, both suffer (the line ping-pongs between cores), and the lock's queuing can actually win because it

stops cores from spinning. Profile the contended case, not just the uncontended one.

- **False sharing.** Counters that are hot from many cores should not share a cache line with other hot fields, or the cache-coherence traffic dominates. Pad or separate truly hot atomics; this is a JIT/CPU-layout concern that microbenchmarks expose.
- **GC and allocation.** A `Lock` is one reference-type allocation per protected resource, created once.
`Parallel.ForEachAsync` and PLINQ allocate partitioners, state machines, and (for PLINQ ordering) buffers — fine for coarse work, wasteful for tiny work. Lambda captures in parallel bodies allocate a closure per call site, not per iteration, but capturing a mutable loop variable into a parallel body is both an allocation and a correctness trap.
- **Async concurrency vs true parallelism — the central decision.** If your work *waits* (network, disk, database), you want **async concurrency**: `await`, `Parallel.ForEachAsync`, `Channel<T>`. It overlaps waiting without consuming threads, and a `Lock` is the wrong tool inside it (use `SemaphoreSlim`). If your work *computes* (parsing, math, image processing), you want **true parallelism**: `Parallel.For` / `ForEach` or PLINQ to spread CPU across cores, with `Interlocked` / `Lock` for the rare shared write. Mixing them — blocking a pool thread inside a parallel loop on an async call's `.Result` — gives you the costs of both and the benefits of neither, and can deadlock the pool. Decide first whether the bottleneck is *waiting* or *computing*, and pick the model that matches.

The distinction is not academic; it determines how many “in-flight” units you can afford. Async concurrency scales to thousands of outstanding operations because each waiting operation costs a small state machine on the heap, not a thread — so `MaxDegreeOfParallelism = 200` for HTTP calls is reasonable, bounded by the server and your sockets, not by cores. True parallelism scales to roughly the core count because each running unit needs a CPU; `MaxDegreeOfParallelism = 200` for a CPU-

bound `Parallel.For` on a 16-core box just oversubscribes the scheduler, causing context-switch thrash and cache eviction that make it *slower* than `MaxDegreeOfParallelism = 16`. The numbers that are sensible in one model are pathological in the other, which is the surest sign they are genuinely different tools. A useful litmus test before writing any concurrent code: if you removed all the synchronization and ran the work on a single thread, would the bottleneck be the CPU pegged at 100% (you want parallelism) or the CPU mostly idle while you wait on something external (you want async)? The answer dictates everything downstream — the primitive, the degree, the collection type, and whether a `Lock` or a `SemaphoreSlim` guards your shared state.

The throughline of this chapter is restraint. The new `Lock` type makes the common case faster and harder to misuse, `Interlocked` removes locks entirely from single-field updates, and `Parallel` / PLINQ scale coarse work across cores — but each is correct only within its lane. The fastest synchronization is the synchronization you designed away by giving each worker its own state and merging once at the end.

Part VI — Metaprogramming and Interop

21 · Reflection, Metadata, and Source Generators

Reflection is the runtime's window into its own type system: powerful, expensive, and increasingly at odds with trimming and Native AOT. This chapter establishes a cost model for reflection, shows how to claw back most of the price with cached compiled delegates, and then moves the work to where it belongs in modern .NET — build time — via Roslyn incremental source generators and analyzers. The throughline is a single decision: resolve metadata once and cache it, or eliminate the runtime resolution entirely by generating code.

The reflection cost model

Reflection is not one cost; it is a stack of them. Acquiring a `Type` is cheap (`typeof(T)` is effectively a load of a runtime handle; `obj.GetType()` is a single indirection through the object header). The expense begins when you ask the type for its members. Every call to `Type.GetMethod`, `GetProperty`, `GetField`, or `GetCustomAttributes` walks metadata tables, allocates `MemberInfo` wrapper objects, and — for the parameterized overloads — runs binder logic to disambiguate overloads.

Three distinct costs deserve separate accounting:

1. **Member lookup** — `GetMethod("Foo")` searches the type's metadata. The returned `MethodInfo` is cached internally by the runtime per `(Type, member)` after the first resolution, but the *lookup call itself* still allocates and does work each time you call it. You must cache the `MemberInfo` yourself.
2. **Attribute materialization** — `GetCustomAttributes` constructs attribute *instances* by invoking their constructors and setting properties. This allocates and runs code every call unless you cache. The `CustomAttributeData` API (`GetCustomAttributesData`) reads the attribute *blob* with-

out constructing instances — cheaper and trim-safe, but you read constructor arguments positionally.

3. **Dynamic invocation** — `MethodInfo.Invoke` and `PropertyInfo.GetValue/SetValue` are the most expensive. Each call boxes value-type arguments into an `object[]`, performs an access check, validates argument types and counts, and dispatches through a slow stub. A single `Invoke` is on the order of hundreds of nanoseconds versus a few nanoseconds for a virtual call.

```
using System.Reflection;

// Cheap: handle load, no metadata walk.
Type t = typeof(Order);

// Lookup cost – do this ONCE and store the result.
MethodInfo? total =
    t.GetMethod(nameof(Order.ComputeTotal),
        BindingFlags.Public | BindingFlags.Instance);

// Invocation cost – boxing + binder + access check on
// EVERY call.
object? result = total!.Invoke(order, parameters:
    null);
```

The cardinal rule: **member lookup is amortizable, invocation is not**. Caching a `MethodInfo` in a `static readonly` field removes the lookup cost permanently, but `Invoke` stays slow no matter how many times you call it. The fix for hot-path invocation is a compiled delegate, covered below.

A `ConcurrentDictionary<Type, T>` keyed on `Type` is the canonical cache for per-type reflection results. `Type` has reference identity per runtime type, so it is a correct and fast dictionary key. Prefer `GetOrAdd` with a static lambda (or the factory overload) to avoid closure allocations on the hot path.

Attributes: defining, retrieving, `AttributeUsage`

An attribute is a class deriving from `System.Attribute` whose constructor arguments and writable properties become compile-time metadata. `AttributeUsage` constrains where it may ap-

pear, whether it may repeat, and whether derived members inherit it — and the compiler enforces all three.

```
// AttributeTargets controls placement; AllowMultiple
// and Inherited are enforced
// by the compiler and by
// Attribute.GetCustomAttributes(inherit:)
// respectively.
[AttributeUsage(AttributeTargets.Class |
    AttributeTargets.Struct,
    AllowMultiple = false, Inherited = false)]
public sealed class GenerateToStringAttribute :
    Attribute
{
    // Positional argument -> constructor parameter.
    public GenerateToStringAttribute(string prefix) =>
        Prefix = prefix;

    public string Prefix { get; }

    // Named argument -> settable property:
    // [GenerateToString("Order", Multiline = true)]
    public bool Multiline { get; init; }
}
```

Two retrieval strategies, with very different cost and trim characteristics:

```

// Strategy A: materialize the instance. Runs the
// attribute ctor; allocates.
var attr =
    type.GetCustomAttribute<GenerateToStringAttrib
        ute>();
bool multiline = attr?.Multiline ?? false;

// Strategy B: read the blob without constructing. No
// ctor call, trim-friendly,
// but you index arguments positionally. Good for
// high-volume scanning.
foreach (CustomAttributeData data in
    type.GetCustomAttributesData())
{
    if (data.AttributeType ==
        typeof(GenerateToStringAttribute))
    {
        string prefix =
            (string)data.ConstructorArguments[0].Value!;
        bool ml = data.NamedArguments
            .FirstOrDefault(n => n.MemberName ==
                nameof(GenerateToStringAttribute.Multiline))
            .TypedValue.Value is true;
    }
}

```

`Inherited = false` matters for both correctness and performance: when `false`, the runtime does not walk the base-type chain looking for the attribute, which is the common case for marker attributes that a generator reads.

Dynamic invocation vs compiled delegates vs source generation

These three are points on a spectrum of *when* you pay and *how much*:

Technique	Cost paid at	Per-call cost	Trim/AOT safe
<code>MethodInfo.Invoke</code>	Runtime, every call	Very high (boxing, binder, access check)	Fragile
Compiled delegate (<code>CreateDelegate</code> / expression tree)	Runtime, once	Near-direct call after warmup	Expression trees need JIT; <code>CreateDelegate</code> is fine
Source generation	Build time	Zero — it is hand-written-equivalent IL	Fully safe

The progression of this chapter is to move left-to-right: cache the `MethodInfo`, then cache a compiled delegate from it, then eliminate reflection altogether with a generator.

Caching a compiled delegate via `CreateDelegate`

`MethodInfo.CreateDelegate` binds a method into a strongly-typed delegate **once**, after which calls go through the delegate's invocation stub at near-virtual-call speed. This is dramatically faster than `Invoke` and avoids the `object[]` boxing entirely when the delegate signature is concrete.


```

using System.Reflection;

public static class Accessor<TTarget, TResult>
{
    // Build a Func<TTarget, TResult> from a
    // parameterless instance method, ONCE.
    public static Func<TTarget, TResult>
        CompileGetter(string methodName)
    {
        MethodInfo mi =
            typeof(TTarget).GetMethod(methodName,
                BindingFlags.Public |
                BindingFlags.Instance)
            ?? throw new
                MissingMethodException(typeof(TTarget).FullName,
                    methodName);

        // Open-instance delegate: first delegate arg
        // becomes the receiver.
        return mi.CreateDelegate<Func<TTarget,
            TResult>>();
    }
}

// Usage: resolve + compile once, then call millions
// of times for free.
var getTotal = Accessor<Order,
    decimal>.CompileGetter(nameof(Order.ComputeTotal));
decimal d = getTotal(order); // ~1-2 ns vs ~250 ns for
    MethodInfo.Invoke

```

The generic `CreateDelegate<TDelegate>()` overload (a clean addition that replaced the older `CreateDelegate(Type, ...)` cast dance) returns a typed delegate directly. For property access where you need `object`-typed getters/setters over many heterogeneous types — the classic serializer scenario — build a `Func<object, object?>` with `System.Linq.Expressions` and cache it per `PropertyInfo`:

```

using System.Linq.Expressions;
using System.Reflection;

static Func<object, object?>
    CompilePropertyGetter(PropertyInfo prop)
{
    // (object instance) => (object)
    // ((TDeclaring)instance).Prop
    ParameterExpression instance =
        Expression.Parameter(typeof(object),
            "instance");
    UnaryExpression typed =
        Expression.Convert(instance,
            prop.DeclaringType!);
    MemberExpression access =
        Expression.Property(typed, prop);
    UnaryExpression boxed = Expression.Convert(access,
        typeof(object));
    return Expression.Lambda<Func<object, object?>>
        (boxed, instance).Compile();
}

```

Caveat: `Expression.Compile()` emits IL at runtime and therefore requires a JIT — it does **not** work under Native AOT (it throws or falls back to a slow interpreter). `CreateDelegate` over an existing `MethodInfo` is AOT-compatible because no new IL is generated. If your destination is AOT, prefer `CreateDelegate`, and if you need cross-type `object` accessors under AOT, you have arrived at the real reason for source generators: generate the accessor at build time.

Benchmark: Invoke vs CreateDelegate

```
using BenchmarkDotNet.Attributes;
using System.Reflection;

[MemoryDiagnoser]
public class InvocationBenchmark
{
    private readonly Order _order = new(qty: 3, unit:
        19.99m);
    private static readonly MethodInfo _mi =
        typeof(Order).GetMethod(nameof(Order.ComputeTotal));
    private static readonly Func<Order, decimal> _del
        =
            _mi.CreateDelegate<Func<Order, decimal>>();

    [Benchmark(Baseline = true)]
    public decimal Direct() => _order.ComputeTotal();

    [Benchmark]
    public decimal ViaDelegate() => _del(_order);

    [Benchmark]
    public decimal ViaInvoke() =>
        (decimal)_mi.Invoke(_order, null!);
}

// Representative results (.NET 10, x64):
// | Method          | Mean | Ratio | Allocated |
// |-----|-----:|-----:|-----:|
// | Direct          | 0.512 ns | 1.00 | - |
// | ViaDelegate     | 1.380 ns | 2.70 | - |
// |   <- cached delegate
// | ViaInvoke       | 243.700 ns | ~476 | 24 B |
// |   <- boxing the decimal return
```

The delegate is within ~1 ns of a direct call and allocates nothing; `Invoke` is two-plus orders of magnitude slower and allocates a box for the `decimal` return. **Re-resolving the `MethodInfo` and calling `Invoke` on a hot path is the single most common reflection performance bug** — it shows up as steady-state allocation and CPU in profilers for serializers, mappers, and dependency-injection containers that never cached.

Roslyn incremental source generators, end to end

Source generators run inside the compiler. An

`IIncrementalGenerator` (the modern API; the legacy `ISourceGenerator` is effectively deprecated for new work) participates in the build by registering a *pipeline* of transforms over the compilation, ending in source outputs that the compiler treats as additional files. The “incremental” contract is the whole game: the driver caches the output of every pipeline stage and re-runs a stage only when its input changes by value equality. Done right, typing a character in an unrelated file re-runs almost nothing; done wrong, it re-runs everything on every keystroke and tanks the IDE.

A generator project targets `netstandard2.0` (the Roslyn host runs on it), references `Microsoft.CodeAnalysis.CSharp`, and is referenced by consumers as an analyzer:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <LangVersion>latest</LangVersion>

    <EnforceExtendedAnalyzerRules>true</EnforceExtendedAnalyzerRules>
    <IncludeBuildOutput>>false</IncludeBuildOutput>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.CodeAnalysis.CSharp"
      Version="4.14.0"
      PrivateAssets="all" />
  </ItemGroup>
</Project>
```

```
<!-- In the CONSUMER project -->
<ItemGroup>
  <ProjectReference
    Include="..\Generators\Generators.csproj"
    OutputItemType="Analyzer"
    ReferenceOutputAssembly="false" />
</ItemGroup>
```

Worked example: a ToString/mapping generator from a marker attribute

This generator scans for `[GenerateToString]` on partial classes and emits a `ToString()` override that concatenates all public properties. It demonstrates the full, correct pipeline shape: inject the marker via `RegisterPostInitializationOutput`, discover with `ForAttributeWithMetadataName`, project into a **small, value-equatable model**, and emit.

```

using System.Collections.Immutable;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp.Syntax;

[Generator(LanguageNames.CSharp)]
public sealed class ToStringGenerator :
    IIncrementalGenerator
{
    private const string AttributeName =
        "Demo.GenerateToStringAttribute";

    public void
        Initialize(IncrementalGeneratorInitializationContext context)
    {
        // 1) Emit the marker attribute itself so
        consumers need no extra package.

        context.RegisterPostInitializationOutput(static
            ctx => ctx.AddSource(
                "GenerateToStringAttribute.g.cs", """
                // <auto-generated>
                #nullable enable
                namespace Demo
                {

                    [System.AttributeUsage(System.AttributeTargets
                        .Class)]
                    internal sealed class
                    GenerateToStringAttribute : System.Attribute {
                }

                """));

        // 2) ForAttributeWithMetadataName: ~99x
        cheaper than CreateSyntaxProvider
        // (Andrew Lock's measured figure) because
        the compiler pre-indexes which
        // syntax trees carry an attribute of this
        metadata name and only runs the
        // predicate against those candidates,
        instead of every node in every tree.
        IncrementalValuesProvider<Model> models =
            context.SyntaxProvider
                .ForAttributeWithMetadataName(
                    AttributeName,
                    predicate: static (node, _) => node is
                        ClassDeclarationSyntax,
                    transform: static (ctx, _) =>
                        Extract(ctx));
    }
}

```

```

// 3) Emit. RegisterSourceOutput so the IDE
// sees the generated members.
context.RegisterSourceOutput(models, static
(spc, model) => Emit(spc, model));
}

// The model is a record of PRIMITIVES ONLY. Never
// store ISymbol/Compilation/
// SyntaxNode here – they are not value-equatable
// and they pin huge object
// graphs, destroying incrementality.
private sealed record Model(
    string Namespace,
    string TypeName,
    EquatableArray<string> PropertyNames);

private static Model
    Extract(GeneratorAttributeSyntaxContext ctx)
{
    var symbol =
        (INamedTypeSymbol)ctx.TargetSymbol;

    var props = symbol.GetMembers()
        .OfType<IPropertySymbol>()
        .Where(p => p.DeclaredAccessibility ==
        Accessibility.Public
            && p.GetMethod is not null &&
            !p.IsStatic)
        .Select(p => p.Name)
        .ToImmutableArray();

    string ns =
        symbol.ContainingNamespace.IsGlobalNamespace
            ? ""
            :
            symbol.ContainingNamespace.ToString();

    return new Model(ns, symbol.Name, new
        EquatableArray<string>(props));
}

private static void Emit(SourceProductionContext
    spc, Model m)
{
    var sb = new StringBuilder();
    sb.AppendLine("// <auto-
    generated/>").AppendLine("#nullable enable");
    bool hasNs = m.Namespace.Length > 0;
    if (hasNs) sb.AppendLine($"namespace
    {m.Namespace}").AppendLine("{");

    sb.AppendLine($"    partial class
    {m.TypeName}")

```

```

        .AppendLine("    {")
        .AppendLine("        public override string
ToString()")
        .AppendLine("    {");

    if (m.PropertyNames.Count == 0)
        sb.AppendLine($"        return \"
{m.TypeName} {{ }}\";");
    else
    {
        sb.Append($"        return $\"
{m.TypeName} {{{ }}\";");
        for (int i = 0; i < m.PropertyNames.Count;
i++)
        {
            string p = m.PropertyNames[i];
            if (i > 0) sb.Append(", ");
            sb.Append($"{{p}} = {{{p}}}");
        }
        sb.AppendLine("    }}\";");
    }

    sb.AppendLine("    }").AppendLine("
}");
    if (hasNs) sb.AppendLine("");

    spc.AddSource($"\"{m.TypeName}.ToString.g.cs\"",
sb.ToString());
}
}

```

`EquatableArray<T>` is the well-known pattern that wraps `ImmutableArray<T>` (whose default `Equals` is reference equality and therefore breaks incrementality) to provide structural equality:


```

using System.Collections;
using System.Collections.Immutable;

public readonly struct EquatableArray<T>
    (ImmutableArray<T> array)
    : IEquatable<EquatableArray<T>>, IEnumerable<T>
    where T : IEquatable<T>
{
    private readonly ImmutableArray<T> _array = array;
    public int Count => _array.IsDefault ? 0 :
        _array.Length;
    public T this[int i] => _array[i];

    public bool Equals(EquatableArray<T> other) =>
        _array.AsSpan()
            .SequenceEqual(other._array.AsSpan());
    public override bool Equals(object? o) => o is
        EquatableArray<T> e && Equals(e);
    public override int GetHashCode()
    {
        var hc = new HashCode();
        foreach (var item in _array.IsDefault ? [] :
            _array) hc.Add(item);
        return hc.ToHashCode();
    }
    public IEnumerator<T> GetEnumerator() =>
        (_array.IsDefault ? Enumerable.Empty<T>() :
            _array).GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() =>
        GetEnumerator();
}

```

Consumer code stays declarative; the partial half is generated:

```

namespace Demo;

[GenerateToString]
public partial class Order
{
    public int Quantity { get; init; }
    public decimal UnitPrice { get; init; }
}
// Generated: public override string ToString()
//             => $"Order {{ Quantity = {Quantity},
//               UnitPrice = {UnitPrice} }}";

```

Note the collection expression `[]` used as an empty-collection fallback in `GetHashCode` above — in that ternary it is target-

typed to `ImmutableArray<T>` (the type of the other branch), producing an empty immutable array. That's the C# 12 collection-expression syntax doing real work inside generator infrastructure.

Reading the semantic model and building syntax

The two ways to produce generated text are *string building* (as above) and the *syntax factory* (`SyntaxFactory / CompilationUnit().WithMembers(...)`). Strings win for generators in practice: they are far easier to read, faster to produce, and don't require constructing and normalizing a syntax tree only to call `.ToString()` on it anyway. Reserve `SyntaxFactory` for analyzers' code fixes, where you must splice into a user's *existing* tree and preserve trivia.

What you read *from* the model, however, should always go through the **semantic model** (`ISymbol`), never raw syntax. Syntax tells you a property is named `Foo` ; only the symbol tells you its accessibility, type, whether it's static, and its fully-qualified namespace across `using` aliases and partial declarations. The `GeneratorAttributeSyntaxContext` hands you `TargetSymbol` and `SemanticModel` directly — use them, then immediately project to primitives so nothing from the compilation survives into the cached pipeline stage.

Pairing generators with partial properties and partial members

C# 13 added **partial properties and indexers** (the declaring half has no body; the implementing half does), and C# 14 added **partial constructors and partial events**, plus the `field` contextual keyword. Together these let a generator supply implementations while the user's declaration stays clean — no `static` factory method names, no separate generated type, just a property that *looks* normal at the call site.

Pre-C# 13, a generator that wanted to back a property had to emit a whole partial class with its own backing field and a full property. With partial properties the user writes the declaration

and the generator writes only the accessor body. With the C# 14 `field` keyword the generator doesn't even need to declare a backing field — `field` binds to the compiler-synthesized backing field, so the field is invisible and inaccessible outside the property, which is exactly what you want for generated lazy/validated/notifying properties.

```
// USER code – declarative. The partial property has  
    NO body here.  
namespace Demo;  
  
public partial class Settings  
{  
    [LazyFromConfig("DOTNET_GCHeapHardLimit")]  
    public partial string? HeapLimit { get; }  
}
```

```
// GENERATOR output – implementing half. C# 14 `field`  
    backs the cache with an  
    // invisible, inaccessible field; no separate backing-  
    field declaration needed.  
namespace Demo;  
  
public partial class Settings  
{  
    public partial string? HeapLimit  
        => field ??=  
            System.Environment.GetEnvironmentVariable("DOT  
                NET_GCHeapHardLimit");  
}
```

The generator side projects the marker's argument into the model and emits the implementing declaration:

```

private static void
    EmitLazyProperty(SourceProductionContext spc,
        LazyModel m)
{
    string src = $$""
        // <auto-generated/>
        #nullable enable
        namespace {{m.Namespace}}
        {
            partial class {{m.TypeName}}
            {
                public partial string?
                    {{m.PropertyName}}
                    => field ??=
                    System.Environment.GetEnvironmentVariable("
                        {{m.EnvVar}}");
            }
        }
    """;
    spc.AddSource($"{{m.TypeName}}.
        {{m.PropertyName}}.g.cs", src);
}

```

The signature on both halves must match exactly, including nullability and accessor list. A missing implementing half is [CS9248](#), and a signature mismatch between the two halves is its own hard compile error in the partial-member diagnostic family — either way the build fails, which is a feature: the generator and the user declaration cannot silently drift. The same pattern applies to partial events (C# 14) for generated [INotifyPropertyChanged](#) plumbing and to partial constructors (C# 14) for generated initialization that the user opts into by declaring [public partial Settings\(\);](#).

Roslyn analyzers and code fixes

An analyzer is a [DiagnosticAnalyzer](#) that inspects the compilation and reports [Diagnostic](#)s; a paired [CodeFixProvider](#) offers an edit. The worked example below flags [lock\(this\)](#) and [lock\(typeof\(X\)\)](#) — both anti-patterns because the lock object is publicly reachable, so unrelated code can deadlock you — and offers a fix that introduces a private lock object. Under C# 13+ the idiomatic fix introduces a [private readonly](#)

`System.Threading.Lock` field (the new dedicated `Lock` type with its `EnterScope()` pattern), not a bare `object` .

```

using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;

[DiagnosticAnalyzer(LanguageNames.CSharp)]
public sealed class LockOnThisAnalyzer :
    DiagnosticAnalyzer
{
    public const string DiagnosticId = "DEM0001";

    private static readonly DiagnosticDescriptor Rule
        = new(
            DiagnosticId,
            title: "Do not lock on 'this' or a type",
            messageFormat: "Lock on a private dedicated
            lock object, not '{0}'",
            category: "Concurrency",
            DiagnosticSeverity.Warning,
            isEnabledByDefault: true);

    public override
        ImmutableArray<DiagnosticDescriptor>
        SupportedDiagnostics => [Rule];

    public override void Initialize(AnalysisContext
        context)
    {
        context.ConfigureGeneratedCodeAnalysis(Generat
            edCodeAnalysisFlags.None);
        context.EnableConcurrentExecution();
        context.RegisterSyntaxNodeAction(AnalyzeLock,
            SyntaxKind.LockStatement);
    }

    private static void
        AnalyzeLock(SyntaxNodeAnalysisContext ctx)
    {
        var stmt = (LockStatementSyntax)ctx.Node;
        ExpressionSyntax expr = stmt.Expression;

        bool isBad = expr is ThisExpressionSyntax
            || expr is TypeOfExpressionSyntax;

        if (isBad)
            ctx.ReportDiagnostic(Diagnostic.Create(
                Rule, expr.GetLocation(),
                expr.ToString()));
    }
}

```

```
}  
}
```

The code fix locates the containing type, inserts a `private readonly Lock _lock = new();` field if absent, and rewrites the `lock` expression to reference it:

```

using System.Collections.Immutable;
using System.Composition;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

[ExportCodeFixProvider(LanguageNames.CSharp), Shared]
public sealed class LockOnThisCodeFix :
    CodeFixProvider
{
    public override ImmutableArray<string>
        FixableDiagnosticIds =>
        [LockOnThisAnalyzer.DiagnosticId];

    public override FixAllProvider GetFixAllProvider()
        =>
        WellKnownFixAllProviders.BatchFixer;

    public override async Task
        RegisterCodeFixesAsync(CodeFixContext context)
    {
        SyntaxNode root = (await
            context.Document.GetSyntaxRootAsync(
                context.CancellationToken))!;
        Diagnostic diag = context.Diagnostics[0];
        var lockStmt =
            root.FindNode(diag.Location.SourceSpan)
                .FirstAncestorOrSelf<LockStatementSyntax>
                ();
        if (lockStmt is null) return;

        context.RegisterCodeFix(
            CodeAction.Create(
                "Introduce private Lock field",
                ct =>
                    IntroduceLockAsync(context.Document, root,
                        lockStmt, ct),
                equivalenceKey: "IntroduceLock"),
            diag);
    }

    private static Task<Document> IntroduceLockAsync(
        Document doc, SyntaxNode root,
        LockStatementSyntax lockStmt,
        CancellationToken ct)
    {
        var type =
            lockStmt.FirstAncestorOrSelf<TypeDeclarationSy
            ntax>()!;
    }

```



```

const string fieldName = "_lock";

// System.Threading.Lock (C# 13) – value-
// typed, EnterScope() pattern aware.
FieldDeclarationSyntax field =
    SyntaxFactory.FieldDeclaration(
        SyntaxFactory.VariableDeclaration(

            SyntaxFactory.ParseTypeName("System.Threading.
            Lock"))

            .AddVariables(SyntaxFactory.VariableDeclarator
            (fieldName)

            .WithInitializer(SyntaxFactory.EqualsValueClau
            se(

                SyntaxFactory.ParseExpression("new()"))))

            .AddModifiers(

                SyntaxFactory.Token(SyntaxKind.PrivateKeyword)
                ,

                SyntaxFactory.Token(SyntaxKind.ReadOnlyKeyword)
                ))

            .WithAdditionalAnnotations(Formatter.Annotation);

LockStatementSyntax newLock =
    lockStmt.WithExpression(
        SyntaxFactory.IdentifierName(fieldName));

bool hasField = type.Members
    .OfType<FieldDeclarationSyntax>()
    .Any(f => f.Declaration.Variables
        .Any(v => v.Identifier.Text ==
        fieldName));

TypeDeclarationSyntax newType =
    type.ReplaceNode(lockStmt, newLock);
if (!hasField)
    newType =
        newType.WithMembers(newType.Members.Insert(0,
        field));

return
    Task.FromResult(doc.WithSyntaxRoot(root.Replace
        eNode(type, newType)));
}
}

```

Because `lock` over a `System.Threading.Lock` field automatically uses the C# 13 `EnterScope()` lock pattern (the compiler recognizes the type and lowers `lock (l) { }` to `using (l.EnterScope()) { }`), the fix needs only to swap the target expression — no statement rewriting. This is a clean illustration of how a new BCL type plus a compiler pattern simplifies a code fix that would otherwise have to rewrite the lock body.

Testing generators and analyzers with the Roslyn harness

The

`Microsoft.CodeAnalysis.CSharp.SourceGenerators.Testing.*` and `Microsoft.CodeAnalysis.Analyzer.Testing.*` packages provide verifier base classes that compile your input, run the driver, and compare against expected generated sources or diagnostics. For generators, populate `TestState.Sources` with input and `TestState.GeneratedSources` with the expected output (keyed by the generator type and hint name), then `RunAsync()`.

```

using Microsoft.CodeAnalysis.CSharp.Testing;
using Microsoft.CodeAnalysis.Testing;
using Xunit;

public class ToStringGeneratorTests
{
    [Fact]
    public async Task
        Emits_ToString_For_Public_Properties()
    {
        const string input = """
            using Demo;
            namespace App;
            [GenerateToString]
            public partial class Order
            {
                public int Quantity { get; init; }
                public decimal UnitPrice { get; init; }
            }
            """;

        const string expected = """
            // <auto-generated>
            #nullable enable
            namespace App
            {
                partial class Order
                {
                    public override string ToString()
                    {
                        return $"Order {{ Quantity =
{Quantity}, UnitPrice = {UnitPrice} }}";
                    }
                }
            }
            """;

        var test = new
            CSharpSourceGeneratorTest<ToStringGenerator,
            DefaultVerifier>
        {
            TestState =
            {
                Sources = { input },
                // hint name must match the AddSource
                hint exactly.
                GeneratedSources =
                {
                    (typeof(ToStringGenerator),

```

```

        "Order.ToString.g.cs",
        expected),
    },
};
// The generator also emits the marker
// attribute via post-init; declare it
// so the verifier doesn't complain about the
// extra generated file.
test.TestState.GeneratedSources.Add(
    (typeof(ToStringGenerator),
    "GenerateToStringAttribute.g.cs",
    /* attribute source */ AttributeSource));

await test.RunAsync();
}

private const string AttributeSource = /* ...the
post-init text... */ "";
}

```

The exact-match nature of `GeneratedSources` is a double-edged sword: it catches every drift, but it also breaks on white-space. Two robustness moves: normalize aggressively in `Emit` (consistent indentation, `\n` line endings — the harness normalizes line endings but not internal whitespace), and for large outputs prefer **snapshot/approval testing** (Verify) which writes a `.received.txt` / `.verified.txt` pair you diff and accept, instead of inlining the expected text. Snapshot tests are far easier to maintain when output legitimately evolves.

For analyzers, the verifier marks expected diagnostics inline with `{|DEMO001:...|}` spans and can apply the code fix and compare the fixed document:

```

using Microsoft.CodeAnalysis.CSharp.Testing;
using Microsoft.CodeAnalysis.Testing;
using Xunit;

public class LockOnThisTests
{
    [Fact]
    public async Task Flags_And_Fixes_LockOnThis()
    {
        const string before = """
            public class C
            {
                public void M() { lock
                ({|DEMO001:this|}) { } }
            }
            """;

        const string after = """
            public class C
            {
                private readonly System.Threading.Lock
                _lock = new();
                public void M() { lock (_lock) { } }
            }
            """;

        await new CSharpCodeFixTest<
            LockOnThisAnalyzer, LockOnThisCodeFix,
            DefaultVerifier>
        {
            TestCode = before,
            FixedCode = after,
        }.RunAsync();
    }
}

```

A second, deeper generator test asserts **incrementality** rather than output: run the driver twice with an unrelated edit and assert that the steps were `Cached` / `Unchanged`, proving your model is value-equatable. This is the test that catches the worst generator bug before it reaches users:

```

[Fact]
public void Pipeline_Is_Cached_On_Unrelated_Edit()
{
    Compilation comp = CreateCompilation(/* source
        with one [GenerateToString] type */);

    // trackIncrementalGeneratorSteps:true is REQUIRED
    // - without it TrackedSteps is
    // empty and the assertion below passes vacuously
    // (a silent false-positive test).
    var options = new GeneratorDriverOptions(
        disabledOutputs:
            IncrementalGeneratorOutputKind.None,
        trackIncrementalGeneratorSteps: true);
    GeneratorDriver driver =
        CSharpGeneratorDriver.Create(
            new[] { new
                ToStringGenerator().AsSourceGenerator() },
            driverOptions: options);

    // Baseline run - every step's reason is New here,
    // so we do NOT assert on it;
    // the driver instance returned carries the cached
    // outputs forward.
    driver = driver.RunGenerators(comp);

    // Add an unrelated class -> a new Compilation.
    // The marker type is untouched,
    // so the attribute-discovery and emit steps must
    // reuse cached results.
    Compilation comp2 = comp.AddSyntaxTrees(
        CSharpSyntaxTree.ParseText("class Unrelated {
            }"));
    driver = driver.RunGenerators(comp2);

    GeneratorRunResult result =
        driver.GetRunResult().Results[0];
    // After the second run every tracked step should
    // report Cached/Unchanged,
    // never New - that is the proof the model is
    // value-equatable.
    Assert.All(
        result.TrackedSteps.SelectMany(s => s.Value)
            .SelectMany(r => r.Outputs),
        o => Assert.Contains(o.Reason,
            new[] { IncrementalStepRunReason.Cached,
                IncrementalStepRunReason.Unchanged }));
}

```

`CSharpGeneratorDriver.Create` takes `ISourceGenerator` instances, so an `IIncrementalGenerator` must be wrapped with

`.AsSourceGenerator()` (or passed as `new[] { generator.AsSourceGenerator() }`). The `Unchanged` reason means the step's input changed but its output compared equal; `Cached` means the input itself compared equal and the step was skipped entirely.

AOT/trim-friendly metaprogramming: why generators beat reflection

Trimming and Native AOT statically remove unreferenced code. Reflection defeats static analysis: when you look up a member by string name, the trimmer cannot prove the member is used, so it may remove it — and your code then throws

`MissingMethodException` / `NullReferenceException` **at run-time, in production, only on the trimmed build**. The trimmer emits `IL2026` / `IL2070` -class warnings precisely so this fails loud at build time, but only if you don't suppress them. `Expression.Compile` and `Reflection.Emit` are worse: they require a JIT that Native AOT does not ship, so they fail unconditionally.

Source generators are the structural answer. Because the code is emitted at build time as ordinary C#, the trimmer sees real, statically-analyzable member references — no runtime metadata lookup, nothing to remove by mistake, nothing for the JIT to compile. This is why the modern BCL replaced reflection-based subsystems with generators: `System.Text.Json` source generation (`JsonSerializerContext`), `LoggerMessage` generators, `[GeneratedRegex]` , `[LibraryImport]` (P/Invoke marshalling), and `[ComImport]` /COM interop generation all exist to make those features AOT- and trim-clean. The performance win (no per-call reflection, no boxing) is real but secondary; the *correctness under trimming* is the primary motivation.

The decision rule:

- **One-off, startup-time, app-level reflection** (read config once, build a DI graph once): plain cached reflection is fine; the cost is paid once and amortized.

- **Hot-path dynamic dispatch over a closed set of types** (serializers, mappers, validators): compile and cache a delegate at minimum; generate at build time if you can.
- **Anything that must trim or run under Native AOT:** source generation, full stop. Do not reach for `Expression.Compile`.

Pitfalls, restated

- **Non-incremental shape kills the IDE.** Capturing `Compilation`, `ISymbol`, `SyntaxNode`, or `SemanticModel` in your pipeline model — or projecting to a non-equatable type like raw `ImmutableArray<T>` — forces the driver to re-run on every keystroke and pins large object graphs in memory. Project to primitives in the `transform` callback and wrap collections in `EquatableArray<T>`. Use `RegisterImplementationSourceOutput` for build-only outputs so the IDE skips them.
- **Reflection that breaks silently under trimming/AOT.** String-keyed member lookup is invisible to the trimmer. Annotate entry points with `[DynamicallyAccessedMembers]` / `[RequiresUnreferencedCode]` so the warnings surface, or — far better — replace the reflection with a generator.
- **Re-resolving `MethodInfo` and `Invoke`-ing on hot paths.** This is the reflection antipattern that profilers find again and again. Cache the `MethodInfo` once in a `static readonly` field, then cache a delegate via `CreateDelegate` and call *that*. The 200×+ gap and the per-call boxing allocation both disappear.

The unifying principle of this chapter: metadata work is fixed cost you should pay exactly once — at build time if the BCL gives you a generator, at warmup via a cached compiled delegate if it doesn't, and never per-call via `Invoke`.

22 · Function Pointers, P/Invoke, and Native Interop

Native interop is where the managed safety net ends and you become personally responsible for memory layout, lifetimes, and calling conventions. .NET 10 gives you sharp, zero-overhead tools for the job — source-generated P/Invoke, unmanaged function pointers, `[UnmanagedCallersOnly]`, and a marshalling pipeline you can opt out of entirely — but every one of them will corrupt the heap or crash the process if you get a single field offset or convention wrong. This chapter is about using them correctly and knowing exactly what the JIT and GC are doing underneath.

The two ways to cross the boundary

There are exactly two mechanisms for calling native code: a **P/Invoke stub** (a marshalling thunk the runtime generates for a `static extern / partial` method) and an **unmanaged function pointer** (`delegate* unmanaged<...>`, a raw machine address the JIT calls directly). P/Invoke is for named exports you resolve at load time; function pointers are for callbacks and addresses you obtain at runtime. Both ultimately do the same low-level thing — transition the thread from cooperative GC mode to preemptive, call the target, transition back — but they differ in how the signature is described and how much ceremony surrounds the call.

A third construct, the managed **delegate**, can also be handed to native code (the runtime synthesizes a stub and gives native a pointer to it), but in modern code you should reach for `delegate* unmanaged` instead. Delegates carry an object, a method-table pointer, and an allocation; the function-pointer form is a single pointer-sized value the JIT emits as one indirect `call` (the IL `calli` instruction) straight to the target. The only reason to still use a delegate-to-native-pointer (`Marshal.GetFunctionPointerForDelegate`) is when the call-

back must capture managed state, since a `delegate*` cannot close over anything.

Source-generated P/Invoke with `[LibraryImport]`

For two decades the canonical P/Invoke was `[DllImport]`, which resolves the export and builds the marshalling stub *at runtime* via an IL stub the JIT emits on first call. That has three costs: it is invisible to the AOT compiler (the stub is generated dynamically, so NativeAOT cannot see it), the marshalling logic is opaque and undebuggable, and errors surface only when the call executes. `[LibraryImport]`, introduced in .NET 7 and now the default recommendation in .NET 10, moves all of that to a Roslyn **source generator**: it emits a C# stub you can read, step into, and that NativeAOT trims and compiles ahead of time.

The mechanics: mark the method `static partial`, put it in a `partial` type, and the generator writes the partial implementation. The legacy version:

```
using System.Runtime.InteropServices;

internal static class LegacyNative
{
    [DllImport("libc", EntryPoint = "getpid",
        SetLastError = false)]
    public static extern int getpid();
}
```

The .NET 10 equivalent — note `partial`, no `extern`, and the enclosing class is also `partial`:

```

using System.Runtime.InteropServices;

internal static partial class Native
{
    [LibraryImport("libc")]
    public static partial int getpid();
    // ^ the source generator supplies the body,
    //    which P/Invokes into libc's getpid
}

// Console.WriteLine(Native.getpid()); // e.g. 48213
//    - the OS process id

```

The generator produces a hidden stub roughly like `public static partial int getpid() => __PInvoke();` where `__PInvoke` is a local `[DllImport]`-style declaration with **no marshalling** because `int` is blittable. That last point is the entire design philosophy of `[LibraryImport]`: it shines for blittable signatures and uses *named marshaller types* (not runtime magic) for everything else, so the generated code is explicit. When a parameter requires non-trivial marshalling that the generator cannot handle, it tells you at *compile time* with a diagnostic, instead of throwing `MarshalDirectiveException` at runtime.

String marshalling is now declared, not inferred. The legacy `CharSet` enum is replaced by the `StringMarshalling` property:

```

using System.Runtime.InteropServices;
using System.Runtime.InteropServices.Marshal; //
    Utf8StringMarshaller lives here

internal static partial class NativeStrings
{
    // UTF-8: the generator uses
    System.Runtime.InteropServices.Marshal.Utf
    8StringMarshaller

    [LibraryImport("libc", StringMarshalling =
        StringMarshalling.Utf8)]
    public static partial int system(string command);

    // For one-off control, override per parameter
    with MarshalUsing (.NET 7+):
    [LibraryImport("libc", EntryPoint = "setenv")]
    public static partial int setenv(
        [MarshalUsing(typeof(Utf8StringMarshaller))]
        string name,
        [MarshalUsing(typeof(Utf8StringMarshaller))]
        string value,
        int overwrite);
}

```

`StringMarshalling.Utf8` selects `Utf8StringMarshaller`;
`StringMarshalling.Utf16` selects `Utf16StringMarshaller` (a
 blittable copy, no transcoding). There is no `Ansi` shortcut —
 ANSI is locale-dependent and the modern API deliberately
 makes you opt in via
`[MarshalUsing(typeof(AnsiStringMarshaller))]`. Always
 prefer UTF-8 for POSIX APIs and UTF-16 for Windows `W`
 exports.

`[DefaultDllImportSearchPaths]` and resolution

`[LibraryImport]` honors the same probing as `[DllImport]`,
 including `NativeLibrary.SetDllImportResolver`, which you
 register once to redirect a logical library name (`"libc"`) to a
 platform-specific file. This is the supported way to handle `libc`
 on Linux vs `libSystem` on macOS vs `msvcrt` on Windows from
 a single import declaration — register a resolver in a module ini-
 tializer rather than littering
`RuntimeInformation.IsOSPlatform` checks through your code.

Calling conventions

A calling convention is the contract for *how* arguments land in registers/stack and *who* cleans up. Mismatch it and you get stack corruption that may not crash until a later, unrelated frame returns — the single hardest interop bug to diagnose. Conventions are expressed with `CallConv*` types in `System.Runtime.CompilerServices`. The base conventions the runtime actually supports for unmanaged calls are `CallConvCdecl`, `CallConvStdcall`, and `CallConvThiscall`; `CallConvMemberFunction` is a modifier that selects the instance-member ABI variant (combine it with a base convention, e.g. `delegate* unmanaged[Thiscall, MemberFunction] <...>`). A type named `CallConvFastcall` exists in the namespace, but the CoreCLR runtime does **not** support a fastcall function-pointer ABI — using it throws at call time, so treat it as reserved, not usable. Remember that the convention only matters on Windows x86: on x64, ARM, and ARM64 there is a single platform convention and the `CallConv*` annotation has no effect.

For `[LibraryImport]` you express the convention with `[UnmanagedCallConv]` (preferred over the older `CallingConvention` enum because it composes with modifiers like `CallConvSuppressGCTransition`):

```
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

internal static partial class Fast
{
    // Cdecl + suppress the GC mode transition for a
    // tiny, non-blocking call.
    [LibraryImport("mathlib")]
    [UnmanagedCallConv(CallConvs = new[]
    {
        typeof(CallConvCdecl),
        typeof(CallConvSuppressGCTransition)
    })]
    public static partial double fast_sqrt(double x);
}
```

`CallConvSuppressGCTransition` (the attribute form of `[SuppressGCTransition]`) skips the cooperative→preemptive→cooperative dance. That transition is a handful of instructions plus a memory barrier; eliminating it can roughly halve the overhead of a call that does almost nothing. **Only** use it for functions that are short, never block, never call back into managed code, and never allocate — because while the transition is suppressed the GC cannot suspend that thread, so a long-running native call will stall every collection process-wide. It is a scalpel, not a default.

Unmanaged function pointers vs delegates

`delegate*` is a C# 9 feature that exposes the IL `calli` instruction. The type `delegate* unmanaged[Cdecl]<int, int, int>` reads as “a pointer to an unmanaged Cdecl function taking two `int`s and returning `int`.” It is an unmanaged pointer type, so it lives in `unsafe` code and can be stored in fields, struct members, and local variables. (Like any pointer type it cannot be a generic type argument, so there is no `Span<delegate* ...>`; if you need a contiguous table of them, declare a raw `delegate* unmanaged<...>*` or a struct with named pointer fields.)

Here is the full lifecycle: obtain a native address (here from `dlsym` / `GetProcAddress` via `NativeLibrary`), cast it to the function-pointer type, store it, and invoke it.

```

using System.Runtime.InteropServices;

internal static unsafe class CallbackDemo
{
    // A native library exports: int (*)(int, int) –
    // e.g. a registered comparator.
    private static delegate* unmanaged[Cdecl]<int,
        int, int> s_combine;

    public static void Init(nint moduleHandle)
    {
        nint addr =
            NativeLibrary.GetExport(moduleHandle,
                "combine");
        s_combine = (delegate* unmanaged[Cdecl]<int,
            int, int>)addr;
    }

    public static int Combine(int a, int b)
    {
        if (s_combine is null)
            throw new
                InvalidOperationException("combine not
                bound");
        return s_combine(a, b); // emits an indirect
            `calli` – no delegate, no allocation
    }
}

```

The call `s_combine(a, b)` emits a single indirect call (`calli`) through the stored address. Compare that to a delegate field: a call through `Delegate.Invoke`, which dereferences the delegate object's target and method-pointer fields, plus a null check and a GC-tracked object reference to keep alive. For a hot callback invoked millions of times — a sort comparator, a SIMD kernel dispatch, a per-pixel callback — the function-pointer form is materially faster and produces zero garbage.

The cost is rigidity. The `[Cdecl]` in the type is part of the type identity; you cannot assign a `delegate* unmanaged[Stdcall]<...>` to it. That is a feature: the compiler enforces convention matching that delegates silently get wrong. **Pitfall:** if the native export is actually `__stdcall` and you typed `[Cdecl]`, the program may run correctly on x64 (where `Cdecl` and `Stdcall` coincide) and then corrupt the stack on x86 or ARM where they dif-

fer. Always confirm the convention from the native header, not from “it worked on my machine.”

Exposing managed methods to native:

`[UnmanagedCallersOnly]`

The inverse problem — handing native code a pointer to *your* managed method — is solved by `[UnmanagedCallersOnly]` (introduced in .NET 5). The attribute tells the runtime to generate a native-callable entry point with no managed-calling-convention prologue, so native code can call it directly through a function pointer. The constraints are strict and enforced: the method must be `static`, must have only **blittable** parameters and return type, must not be generic nor live in a generic type, and **must not be called from managed code**.


```

using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

internal static unsafe class Exported
{
    // Native side: typedef int (*log_cb)(int level,
    // const char* msg);
    // We expose a managed callback matching that ABI
    // exactly.
    [UnmanagedCallersOnly(CallConvs = new[] {
        typeof(CallConvCdecl) })]
    private static int OnLog(int level, byte*
        utf8Message)
    {
        // utf8Message is owned by the caller; copy
        // out before returning.
        string msg =
            Marshal.PtrToStringUTF8((nint)utf8Message) ??
            string.Empty;
        Console.WriteLine($"[native:{level}] {msg}");
        return 0; // contract: 0 = handled
    }

    public static void Register(nint module)
    {
        // Take the address of the managed method as
        // an unmanaged function pointer.
        delegate* unmanaged[Cdecl]<int, byte*, int> cb
            = &OnLog;

        var set = (delegate* unmanaged[Cdecl]
            <delegate* unmanaged[Cdecl]<int, byte*, int>,
            void>)
            NativeLibrary.GetExport(module,
                "set_log_callback");
        set(cb);
    }
}

```

The `&OnLog` operator yields the native entry point directly — no delegate, no allocation, and crucially **nothing for the GC to collect**, because there is no managed object backing the pointer. This is the decisive advantage over

`Marshal.GetFunctionPointerForDelegate`. Inside the body you are running managed code with a live GC, so you must marshal arguments by hand (note `Marshal.PtrToStringUTF8`) since the runtime applied no marshalling at the boundary. Exceptions must not escape an `[UnmanagedCallersOnly]`

method — an unhandled exception crossing back into native code tears down the process; wrap the body in `try/catch` and translate failures into a native error code.

SafeHandle: deterministic native cleanup

Every native resource you acquire — a file descriptor, a window handle, a library context — must be released, and a bare `nint` field gives you no protection against double-free, use-after-free, or leak-on-exception. `SafeHandle` is the correct abstraction: a critically-finalizable wrapper that the marshalling layer understands, that guarantees release even on `ThreadAbort`, and that blocks the handle from being recycled while a P/Invoke is mid-flight (the runtime ref-counts it across the call).

Subclass `SafeHandle`, implement `ReleaseHandle` (called exactly once, on a thread the runtime guarantees can run it) and `IsValid`. Pass the subclass type directly as a P/Invoke parameter and the generated stub marshals the underlying handle and manages the ref-count for you.

```

using System.Runtime.InteropServices;

internal static partial class Crypto
{
    [LibraryImport("crypto")]
    public static partial nint ctx_create();

    [LibraryImport("crypto")]
    public static partial void ctx_free(nint ctx);

    [LibraryImport("crypto")]
    public static partial int ctx_update(CryptoContext
        ctx, ReadOnlySpan<byte> data, int len);
    // ^ SafeHandle
    // marshals to the raw nint and is kept alive
    // + ref-counted across the call
}

internal sealed class CryptoContext : SafeHandle
{
    // ownsHandle: true => we are responsible for
    // releasing it.
    private CryptoContext() : base(invalidHandleValue:
        0, ownsHandle: true) { }

    public override bool IsInvalid => handle == 0;

    public static CryptoContext Create()
    {
        var h = new CryptoContext();
        nint raw = Crypto.ctx_create();
        if (raw == 0)
            throw new
                InvalidOperationException("ctx_create
                    failed");
        h.SetHandle(raw); // publish the handle
                          // only after success
        return h;
    }

    // Called once by the runtime; runs even during
    // process shutdown / ThreadAbort.
    protected override bool ReleaseHandle()
    {
        Crypto.ctx_free(handle);
        return true;
    }
}

```

```
// Usage – deterministic disposal, exception-safe,  
//          never double-frees:  
// using var ctx = CryptoContext.Create();  
// Crypto.ctx_update(ctx, payload, payload.Length);
```

Two subtleties. First, `SetHandle` is called *after* the native allocation succeeds, never before — otherwise a partially-constructed handle could leak. Second, never expose the raw `uint` as a property that escapes the `SafeHandle`’s lifetime; the whole point is that the only path to the handle goes through code that increments the ref-count. For descriptor-style integer handles where `0` is valid and `-1` means invalid, derive from `SafeHandleMinusOneIsInvalid` or `SafeHandleZeroOrMinusOneIsInvalid` instead of writing `IsValid` yourself.

Struct layout and inline arrays

When you pass a struct across the boundary, the .NET layout must match the native struct *byte for byte*. Use `[StructLayout(LayoutKind.Sequential)]` (the default for structs, but state it explicitly for interop intent) and, when the native side uses non-default alignment, set `Pack`. A struct is **blittable** — directly copyable with no marshalling — when every field is blittable and the layout is sequential or explicit. Blittable structs are the fast path: the marshaller `memcpy`s them (or pins and passes by reference) with no per-field work.

Fixed-size native fields — `char name[32]`, `uint8_t mac[6]` — are the classic trap. The historical tools were `fixed` buffers (limited to a few primitive types, `unsafe` only) and `[MarshalAs(UnmanagedType.ByValArray, SizeConst = ...)]` (non-blittable, forces field-by-field marshalling). The modern answer is the `[InlineArray]` struct (introduced in C# 12), which gives you a real fixed-size, blittable, embeddable value with `Span<T>` access and no `unsafe` keyword:

```

using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[InlineArray(6)]           // exactly 6
    contiguous bytes, blittable
internal struct MacAddress
{
    private byte _element0;    // the single
        field InlineArray repeats
}

// Native: struct net_iface { uint32_t index; uint8_t
    mac[6]; uint16_t mtu; };
[StructLayout(LayoutKind.Sequential, Pack = 1)]
internal struct NetIface
{
    public uint Index;
    public MacAddress Mac;      // 6 bytes
        inline, no pointer, fully blittable
    public ushort Mtu;
}

internal static partial class Net
{
    [LibraryImport("net")]
    public static partial int get_iface(int index, out
        NetIface iface);
}

// Usage:
// Net.get_iface(0, out NetIface ni);
// ReadOnlySpan<byte> mac = ni.Mac;           // Span
    access for free (C# 12 inline arrays)
// Console.WriteLine($"MTU={ni.Mtu}, MAC=
    {Convert.ToHexStringLower(mac)}");

```

Because `NetIface` is fully blittable, the `out NetIface` marshals as a pinned pointer to the managed struct — zero copying, zero per-field marshalling. Had you modeled `mac` with `[MarshalAs(ByValArray)]`, the struct would become non-blittable and the marshaller would allocate and copy field-by-field on every call. **Pitfall:** mixing a non-blittable field into an otherwise-blittable struct silently demotes the whole struct off the fast path and, worse, if your `Pack` or field order disagrees with the native header by even one byte, you corrupt adjacent memory. Verify offsets with `Marshal.OffsetOf<NetIface>`

`(nameof(NetIface.Mtu))` in a unit test against the known native layout.

Pinning and passing managed memory: `GCHandle`

The GC moves objects during compaction. The instant you hand native code a pointer into a managed array, that pointer can be invalidated by a collection mid-call. Within a single synchronous P/Invoke, the marshaller pins automatically. But when native code must hold a pointer *across* calls — registering a buffer, an async callback that writes into your array later — you must pin explicitly and keep it pinned until native is provably done.

For scoped pinning use `fixed` (cheapest, stack-scoped) or `GCHandle.Alloc(obj, GCHandleType.Pinned)` when the lifetime outlives a lexical block. A pinned `GCHandle` roots the object *and* freezes its address; you **must** `Free()` it or you permanently fragment the heap.

```

using System.Runtime.InteropServices;

// Native: void register_buffer(void* p, int len); //
// keeps p until unregister_buffer()
internal static partial class Dma
{
    [LibraryImport("dma")] public static partial void
        register_buffer(nint p, int len);
    [LibraryImport("dma")] public static partial void
        unregister_buffer(nint p);
}

internal sealed class PinnedBuffer : IDisposable
{
    private GCHandle _handle; // pinned:
    // roots the array AND fixes its address
    public byte[] Data { get; }

    public PinnedBuffer(int size)
    {
        Data = new byte[size];
        _handle = GCHandle.Alloc(Data,
            GCHandleType.Pinned);

        Dma.register_buffer(_handle.AddrOfPinnedObject
            (), size);
    }

    public void Dispose()
    {
        if (_handle.IsAllocated)
        {
            Dma.unregister_buffer(_handle.AddrOfPinnedObje
                ct());
            _handle.Free(); // unpins; the
            // array can move/collect again
        }
    }
}

```

Pinning has a real cost: a pinned object cannot move, so the GC works around it, fragmenting the heap and degrading compaction. Pin late, free early, and for large long-lived buffers prefer the **pinned object heap** (`GC.AllocateArray<byte>(size, pinned: true)`), a dedicated heap segment where allocations are born pinned and don't fragment the regular generations.

Worked example: a native sort with a managed comparator

This ties the chapter together — a P/Invoke into `qsort`, a managed comparator exposed via `[UnmanagedCallersOnly]`, pinning of the managed array, and a function-pointer cast — exactly the shape of a real interop boundary, with every lifetime hazard present.


```

using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

internal static unsafe partial class NativeSort
{
    // void qsort(void* base, size_t nmemb, size_t
    //           size,
    //           int (*compar)(const void*, const
    //           void*));
    [LibraryImport("libc")]
    [UnmanagedCallConv(CallConvs = new[] {
        typeof(CallConvCdecl) })]
    private static partial void qsort(
        void* @base, nuint nmemb, nuint size,
        delegate* unmanaged[Cdecl]<void*, void*, int>
        compar);

    // Managed comparator with the exact C ABI. No
    // managed object backs &Compare,
    // so there is nothing for the GC to collect – the
    // classic delegate-collection
    // bug cannot occur here.
    [UnmanagedCallersOnly(CallConvs = new[] {
        typeof(CallConvCdecl) })]
    private static int Compare(void* a, void* b)
    {
        int x = *(int*)a, y = *(int*)b;
        return x < y ? -1 : (x > y ? 1 : 0);
    }

    public static void Sort(int[] data)
    {
        // qsort returns before we return, so a
        // `fixed` pin is sufficient and cheapest.
        fixed (int* p = data)
        {
            qsort(p, (nuint)data.Length,
                (nuint)sizeof(int), &Compare);
        }
        // pin released here; comparator address was
        // valid for the whole synchronous call
    }
}

// var arr = new[] { 5, 2, 9, 1, 7 };
// NativeSort.Sort(arr);           // arr is now { 1,
//                               2, 5, 7, 9 }

```

Three lifetime facts make this correct. The array is pinned for the entire duration of `qsort`, so the pointers `qsort` dereferences

stay valid. The comparator is an `[UnmanagedCallersOnly]` method, so `&Compare` is a stable native entry point with no GC-managed backing object — there is nothing to root and nothing to collect. And the convention is `Cdecl` on **both** the function-pointer type and the `[UnmanagedCallersOnly]` attribute, so the stack discipline matches.

Now the failure mode this example deliberately avoids. The legacy way to pass a comparator is a managed delegate:

```
// ANTIPATTERN – do not ship this.
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
delegate int CompareFn(IntPtr a, IntPtr b);

CompareFn cmp = (a, b) => 0;
// managed delegate
IntPtr fp =
    Marshal.GetFunctionPointerForDelegate(cmp);
// thunk address
qsortLegacy(p, n, sz, fp);
// If `cmp` is a local with no remaining reference
// after this line, the GC may collect
// it WHILE qsort is still calling through fp ->
// access violation, intermittently,
// usually only under load. Fix: store cmp in a field
// that outlives the native call,
// or call GC.KeepAlive(cmp) AFTER qsortLegacy returns
// (a no-op that simply extends
// the delegate's reachability past the native call).
```

The delegate-collection bug is the most common interop crash in the wild precisely because it is nondeterministic — the delegate is usually still alive by luck, until a GC happens to fire mid-call. The `delegate* unmanaged + [UnmanagedCallersOnly]` pair eliminates the entire class of bug because no collectible object exists.

Diagnosing interop crashes

Interop bugs fall into three buckets, each with a signature:

- **Calling-convention mismatch** — stack corruption that manifests as a crash in a frame *after* the native call returns, or garbage in subsequent locals. The tell is “works on x64,

crashes on x86/ARM.” Audit every `delegate*` and `[UnmanagedCallConv]` against the native header; on x64 Cdecl/Stdcall coincide, so x64-only testing hides the bug.

- **Marshalling/layout mismatch** — heap corruption from a struct whose `Pack`, field order, or fixed-array size disagrees with native by even one byte. Reproduce deterministically by asserting `Marshal.SizeOf<T>()` and `Marshal.OffsetOf<T>(...)` against known-good values in a test. Prefer fully-blittable structs so the marshaller copies rather than reconstructs.
- **Lifetime bug** — use-after-free of a managed object the GC moved or collected (the delegate-collection and unpinned-buffer cases above). These are intermittent and load-dependent. Flush them out with `DOTNET_gcServer=0` `DOTNET_GCStress=0xC` (GC stress mode collects aggressively, turning rare races into reliable failures), and audit every native-held pointer for a rooting story.

No single switch catches all three, so layer the targeted tools: run the lifetime-sensitive paths under `DOTNET_GCStress` to force the rare races; gate every native struct behind `Marshal.SizeOf / OffsetOf` assertions in a test to catch layout drift; and confirm conventions against the native header before you trust the build. For marshalling specifically, `[LibraryImport]` is itself a diagnostic tool — if the source generator refuses to marshal a signature, that is a *compile-time* error about a mismatch that `[DllImport]` would have let through to a runtime crash. Enabling NativeAOT or the trimming/interop analyzers (`<EnableAOTAnalyzer>` / `IL2050` -class warnings) surfaces a further class of unsupported-marshalling patterns at build time rather than at runtime.

When not to reach for interop

Native interop is a liability budget. Every boundary is unverifiable by the type system, untrimmed unless you used `[LibraryImport]`, and a permanent ABI commitment to whatever native version you built against. Before adding a P/Invoke,

check whether the BCL already wraps the syscall (it usually does — `Environment.ProcessId` is `getpid()` without any of the above). Reach for raw interop only when there is no managed surface, when you need a specific native library's algorithms, or when the call is on a hot path where even the managed wrapper's overhead matters. And when you do cross the boundary, prefer the modern stack end to end: `[LibraryImport]` over `[DllImport]`, `delegate* unmanaged` over delegates, `[UnmanagedCallersOnly]` over `GetFunctionPointerForDelegate`, `SafeHandle` over raw `IntPtr`, and `[InlineArray]` over `fixed / ByValArray`. Each modern choice removes an allocation, a GC hazard, or a class of silent corruption.

23 · NativeAOT, Trimming, and Fast Startup

NativeAOT compiles your C# straight to a self-contained native executable — no JIT, no IL on disk, no runtime code generation. The payoff is millisecond cold starts and a small, single-file binary; the price is a permanently closed-world assumption that breaks every pattern depending on reflection-driven discovery or runtime codegen. This chapter is about earning that payoff without shipping a `MissingMethodException` to production.

What NativeAOT actually is

A normal .NET app ships as IL. At process start the runtime maps your assemblies, the JIT compiles methods to machine code on first call, and reflection can inspect or even emit new types on demand. NativeAOT replaces that model entirely. The ILC (IL Compiler) toolchain runs at *publish* time, walks the reachable call graph starting from `Main`, compiles every reachable method to native code ahead of time, and links the result with a minimal runtime (GC, type system, exception handling) into one native binary. There is no JIT in the shipped product and no IL metadata for arbitrary types.

That single design choice cascades into everything else:

- **No runtime code generation.**

`System.Reflection.Emit`, `DynamicMethod`, `Expression.Compile()` (the compiling path), and anything that fabricates IL at runtime cannot work. The compiler that would consume that IL isn't in the binary.

- **Closed world.** ILC must see, at publish time, every type that will ever be instantiated or reflected over.

`Assembly.LoadFrom` of a plugin you compile later, or `Type.GetType("SomeName")` where the name is computed at runtime, has no static answer.

- **Trimming is mandatory.** AOT implies the trimmer. Code ILC can't prove reachable is removed, which is what makes the binary small — and what makes “I’ll reflect over it later” fail.

What you get in return is concrete and measurable: startup dominated by `exec` + page-in rather than JIT warm-up, a working set without JIT scratch buffers or IL metadata, no first-request latency cliff, and a binary you can drop into a `scratch` or `distroless` container with no runtime install. For CLIs, serverless functions, sidecars, and latency-sensitive microservices this is transformative. For a plugin host, a serialization library that reflects over arbitrary user types, or an app leaning on `dynamic`, it is a poor fit — and you should know that *before* you flip the switch, not when a consumer’s publish fails.

Turn it on per project:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <LangVersion>14.0</LangVersion>
    <Nullable>enable</Nullable>

    <!-- Turn on NativeAOT. This implies trimming and
         single-file. -->
    <PublishAot>true</PublishAot>

    <!-- Promote trim/AOT analyzer findings to build
         errors so they
         can never silently reach the publish step. --
    >

    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>

    <!-- Optional size tuning: invariant globalization
         drops ICU. -->

    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>
</Project>
```

`PublishAot` only takes effect on `dotnet publish` with a runtime identifier; ordinary `dotnet build` and `dotnet run` still JIT. Crucially, though, setting `PublishAot` turns on the **trim and AOT analyzers during normal builds**, so you see the warnings at your desk instead of in a consumer's CI.

The trim graph: roots, marks, and warnings

Trimming is reachability analysis. The trimmer starts from a set of **roots** — your entry point, anything referenced by it, and assemblies/types you explicitly preserve — and marks everything statically reachable. Unmarked members are removed. The mental model that matters: *the trimmer only follows edges it can see in IL*. A direct method call is an edge. `typeof(Foo)` followed by `Activator.CreateInstance` is **not** an edge the trimmer understands — it sees a `string / Type` going into a reflection API and has no idea which members you'll pull out.

When the trimmer encounters a reflection pattern it cannot statically resolve, it emits a **trim warning** (the `IL2xxx` family). These are not noise; each one marks a place where the running app may diverge from what got linked. The canonical failure: you reflect over a property the trimmer removed, and you get `MissingMethodException` or a silently-empty result at runtime. The corresponding AOT family is `IL3xxx` — most importantly `IL3050`, “avoid calling members annotated with `RequiresDynamicCode`,” which flags runtime-codegen paths that simply have no implementation under AOT.

Three project knobs control the analysis surface. The umbrella is `IsAotCompatible`:

```
<!-- For a LIBRARY you want to certify as AOT/trim
      safe.
      Setting IsAotCompatible=true implicitly enables
           IsTrimmable,
           EnableTrimAnalyzer, EnableAotAnalyzer, and
           EnableSingleFileAnalyzer. -->
<PropertyGroup>
  <IsAotCompatible>true</IsAotCompatible>
</PropertyGroup>
```

For a library you can't yet make fully AOT-safe but want trimmable, set `<IsTrimmable>true</IsTrimmable>` alone, which marks the assembly `[AssemblyMetadata("IsTrimmable", "True")]` so consumers' trimmers will actually trim into it (rather than preserving it wholesale) and surface warnings against your public surface.

Patterns that break, and why

Three families of code are fundamentally incompatible:

Open-ended reflection.

`Activator.CreateInstance(Type.GetType(name))`, scanning `AppDomain.CurrentDomain.GetAssemblies()` for types implementing an interface, binding to members by string name. The trimmer can't know which types survive; under AOT the metadata may not exist at all.

dynamic. The DLR builds and *compiles* call sites at runtime — that's runtime codegen. `dynamic` is annotated `[RequiresDynamicCode]` end to end and produces `IL3050`. There is no AOT-safe `dynamic`; convert to static dispatch, an interface, or source generation.

Runtime codegen. `Reflection.Emit`, `DynamicMethod`, `Expression.Compile()`, `ImplementInterface`-style proxy generators (Castle DynamicProxy, the EF Core lazy-loading proxies, Moq). All produce `IL3050`. The interpreted expression path `(Expression.Lambda(...).Compile(preferInterpretation: true))` is itself annotated and still warns; treat expression trees as data, not as a JIT.

A representative offender and the analyzer output:


```

using System.Reflection;

public static class PluginLoader
{
    // Under PublishAot this method produces two trim
    // warnings: IL2057 for the
    // runtime-string Type.GetType, and IL2067 for the
    // under-annotated Type
    // flowing into Activator.CreateInstance. Neither
    // is IL2026 – that code only
    // fires when you *call* a
    // [RequiresUnreferencedCode]-annotated member.
    public static object Load(string typeName)
    {
        Type t = Type.GetType(typeName)
            ?? throw new
                InvalidOperationException($"Unknown type
                {typeName}");
        // Trimmer can't see which constructor
        // survives -> may be trimmed away.
        return Activator.CreateInstance(t)!;
    }
}

// Build output (with PublishAot=true and
// TreatWarningsAsErrors):
// error IL2057: Unrecognized value passed to the
// parameter 'typeName' of
// method
// 'System.Type.GetType(String)'. It's not
// possible to
// guarantee the availability of the
// target type.
// error IL2067: 'type' argument does not satisfy
// 'DynamicallyAccessedMemberTypes.PublicParamete
// rlessConstructor'
// in call to
// 'System.Activator.CreateInstance(Type)'.

```

Annotating the public surface: the three attributes

When you can't make a method statically analyzable, your obligation is to **tell the truth in the type system** so the warning propagates to callers instead of vanishing. Three attributes from `System.Diagnostics.CodeAnalysis` do this.

`[RequiresUnreferencedCode]` says “this method uses reflection over members that trimming may remove.” It silences the trim-analysis warnings (`IL2057` , `IL2067` , nested `IL2026` s, ...) *inside* the method body and instead re-raises a single `IL2026` at every call site, forcing callers to opt in or annotate themselves.

`[RequiresDynamicCode]` does the same for `IL3050` / runtime codegen, re-raising `IL3050` at the call site.

`[DynamicallyAccessedMembers]` is the precise alternative: it does *not* suppress warnings — it tells the trimmer exactly which members of a `Type` flowing into a method must be preserved, turning an unanalyzable pattern into an analyzable one.

Here is the idiomatic shape: a trim-*unsafe* convenience overload that is honestly annotated, plus a trim-*safe* sibling that flows the requirement through `[DynamicallyAccessedMembers]` so the trimmer keeps exactly the constructor it needs.

```

using System.Diagnostics.CodeAnalysis;

public static class Factory
{
    // UNSAFE convenience: the type name is a runtime
    // string, so the trimmer
    // cannot preserve the right members. We declare
    // that honestly; the warning
    // now lands on whoever calls Create(string).
    [RequiresUnreferencedCode(
        "Resolves the type by name via reflection; the
        target may be trimmed. " +
        "Use Create<T>() for a trim-safe
        alternative.")]
    public static object Create(string typeName)
    {
        Type t = Type.GetType(typeName, throwOnError:
            true!);
        return Activator.CreateInstance(t!);
    }

    // SAFE alternative. The
    // DynamicallyAccessedMembers annotation tells
    // the
    // trimmer: keep the public parameterless
    // constructor of whatever T is.
    // No IL2026 – the requirement is expressed
    // statically and satisfiable.
    public static T Create<
        [DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicParameterlessConstructor)]
        T>()
        where T : new()
        => new T();
    }
}

```

`[RequiresUnreferencedCode]` (introduced with the linker in .NET 5, hardened across .NET 6–8) is a *contract*, not a fix: it does not make the code work, it makes the danger visible. The discipline is: annotate at the lowest level where reflection is genuinely required, provide a statically-analyzable alternative wherever possible, and never let a `[RequiresUnreferencedCode]` method be reachable from an AOT entry point you intend to ship.

Source generators: the AOT-native replacement for reflection

The strategic move is to push work that *was* done by reflection at runtime to *code generation at compile time*. The generated code is ordinary, statically-analyzable C# the trimmer follows like any other call — no warnings, nothing to suppress, and faster because there's no reflection on the hot path. Three pillars of the BCL ship source generators precisely for this.

JSON: `JsonSerializerContext`

Reflection-based `JsonSerializer.Serialize(obj)` walks the type with reflection and, for `Deserialize`, builds converters dynamically — both `[RequiresUnreferencedCode]` and `[RequiresDynamicCode]`. The source generator (introduced in .NET 6, the standard for AOT since .NET 8) emits the metadata and converters at compile time. You declare a `partial` context deriving from `JsonSerializerContext` and decorate it with `[JsonSerializable]` for each root type.

```

using System.Text.Json;
using System.Text.Json.Serialization;

public sealed record WeatherReport(
    string City, double TempC, string[] Conditions,
    DateOnly Date);

// One partial context lists every root type you
// serialize. The generator
// emits metadata + converters at compile time. The
// default mode emits both
// metadata and a serialize-only fast path;
// JsonSerializerGenerationMode.Serialization
// emits ONLY that write-only fast path (no
// deserialization). Both are AOT-safe.
[JsonSourceGenerationOptions(
    PropertyNamingPolicy =
        JsonKnownNamingPolicy.CamelCase,
    WriteIndented = false)]
[JsonSerializable(typeof(WeatherReport))]
[JsonSerializable(typeof(WeatherReport[]))]
internal partial class AppJsonContext :
    JsonSerializerContext { }

public static class WeatherApi
{
    public static string ToJson(WeatherReport r)
        // Overload taking JsonTypeInfo<T> -> fully
        // reflection-free, AOT-safe.
        => JsonSerializer.Serialize(r,
            AppJsonContext.Default.WeatherReport);

    public static WeatherReport? FromJson(string json)
        => JsonSerializer.Deserialize(json,
            AppJsonContext.Default.WeatherReport);
}

```

The load-bearing detail: call the overloads that take `JsonTypeInfo<T>` or a `JsonSerializerContext`. The plain `Serialize(object)` overloads remain `[RequiresUnreferencedCode]` even when a context exists — passing `AppJsonContext.Default.WeatherReport` is what makes the call statically resolved. Get this wrong and you compile clean but warn under `PublishAot`.

For ASP.NET Core minimal APIs under AOT, register the context so the framework uses it:

```
var builder = WebApplication.CreateSlimBuilder(args);
// slim = AOT-friendly host
builder.Services.ConfigureHttpJsonOptions(o =>
    o.SerializerOptions.TypeInfoResolverChain.Insert(0, AppJsonContext.Default));
```

Regex: [GeneratedRegex]

A runtime `new Regex(pattern)` parses and (for compiled regexes) *emits IL* — `RegexOptions.Compiled` is outright runtime codegen and breaks under AOT, while even interpreted regex pays parse cost at every cold start. The Regex source generator (introduced in .NET 7; targets a `partial` property since .NET 9) emits a fully custom, allocation-lean matcher at compile time. It needs no runtime codegen, satisfies AOT, and moves the pattern-compilation cost to build time.

```
using System.Text.RegularExpressions;

public static partial class Slugs
{
    // BEFORE – runtime construction;
    //   RegexOptions.Compiled is codegen (IL3050),
    // and even without it the pattern is parsed at
    //   startup:
    //   private static readonly Regex _invalid =
    //       new(@"^[a-z0-9]+",
    //         RegexOptions.Compiled);

    // AFTER – generated at compile time. No runtime
    //   parse, no codegen, AOT-safe.
    [GeneratedRegex(@"^[a-z0-9]+",
        RegexOptions.CultureInvariant)]
    private static partial Regex Invalid();

    public static string Slugify(string input) =>
        Invalid().Replace(input.ToLowerInvariant(), "-")
        .Trim('-');
}

// Slugify("Hello, .NET 10!") => "hello-net-10"
```

Configuration binding

`IConfiguration.Get<T>()` and `.Bind(obj)` reflect over `T`'s properties. Set

```
<EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>
```

 and the binder source generator (introduced in .NET 8) intercepts those calls with generated, reflection-free binding code — the AOT-safe path for options binding.

A worked example: a fast CLI from JIT to NativeAOT

This ties the pieces together — JSON, regex, honest annotation — into a small but realistic tool that reads a JSON payload, normalizes names, and prints a report. It is fully AOT-clean.

```

using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

internal static partial class Program
{
    [GeneratedRegex(@"\s{2,}",
        RegexOptions.CultureInvariant)]
    private static partial Regex Whitespace();

    private static int Main(string[] args)
    {
        if (args.Length < 1)
        {
            Console.Error.WriteLine("usage: report
<people.json>");
            return 1;
        }

        string raw = File.ReadAllText(args[0]);

        // Reflection-free deserialize via the
        // generated context.
        Person[]? people =
            JsonSerializer.Deserialize(raw,
                AppJson.Default.PersonArray);
        if (people is null or { Length: 0 })
        {
            Console.Error.WriteLine("no records");
            return 2;
        }

        // C# 14 collection expression + normalization
        // through the generated regex.
        Person[] cleaned =
            [.. people.Select(p => p with { Name =
                Whitespace().Replace(p.Name.Trim(), " ") })];

        string outJson =
            JsonSerializer.Serialize(cleaned,
                AppJson.Default.PersonArray);
        Console.WriteLine($"normalized
{cleaned.Length} records:");
        Console.WriteLine(outJson);
        return 0;
    }
}

internal sealed record Person(string Name, int Age);

```



```
[JsonSourceGenerationOptions(PropertyNamingPolicy =
    JsonKnownNamingPolicy.CamelCase)]
[JsonSerializable(typeof(Person[]))]
internal partial class AppJson : JsonSerializerContext
{ }
```

Nothing here uses reflection on a hot path, `dynamic`, or runtime codegen, so it publishes under `PublishAot` with zero `IL2xxx` / `IL3xxx` warnings. Note the C# 14 ergonomics carried along: the **collection expression with spread** `[.. people.Select(...)]` (collection expressions arrived in C# 12; the spread element is part of that feature) and the non-destructive `with` on a `record`. None of these touch reflection, so they're free under AOT.

Publishing: RIDs, self-contained, platform reality

NativeAOT is always **self-contained** and **platform-specific**. You must publish for a concrete runtime identifier — there is no portable native binary. You cannot cross-compile from arbitrary host to arbitrary target out of the box: build Linux binaries on Linux (or in a container), Windows on Windows, with a matching CPU architecture. The native toolchain (a C/C++ linker — `clang` / `gcc` or the MSVC linker) must be present.

```
# Linux x64
dotnet publish -r linux-x64 -c Release

# Linux Arm64 (e.g. Graviton / Ampere)
dotnet publish -r linux-arm64 -c Release

# Windows x64
dotnet publish -r win-x64 -c Release

# macOS Apple Silicon
dotnet publish -r osx-arm64 -c Release
```

For a container, the natural endpoint is a tiny base image, since there is no .NET runtime to install:

```
# build stage uses the SDK + native toolchain
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
RUN apt-get update && apt-get install -y clang zlib1g-dev
WORKDIR /src
COPY . .
RUN dotnet publish -r linux-x64 -c Release -o /app

# runtime stage: no .NET, just the OS deps the native binary needs
FROM mcr.microsoft.com/dotnet/runtime-deps:10.0-noble-chiseled
WORKDIR /app
COPY --from=build /app/report ./report
ENTRYPOINT ["/report"]
```

.NET 10 expands this story to **tooling**: platform-specific .NET tools can now be packed for NuGet as self-contained, trimmed, or NativeAOT across multiple RIDs, and file-based apps (`dotnet run app.cs`) gained publish and AOT support — so a single-file script can become a native executable. Size tuning beyond the defaults lives in the same csproj:

```
<PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
    <!-- drop ICU -->
    <UseSystemResourceKeys>true</UseSystemResourceKeys>
    <!-- terse exception text -->
    <StackTraceSupport>false</StackTraceSupport>
    <!-- smaller, no managed traces -->
    <OptimizationPreference>Size</OptimizationPreference>
    <!-- vs. Speed -->
</PropertyGroup>
```

Each trades something — `InvariantGlobalization` changes culture-sensitive comparisons and formatting; `StackTraceSupport=false` removes readable managed stack traces. Choose deliberately.

Measuring: size, startup, working set

Don't guess; measure the same app two ways.

```

# JIT, framework-dependent
dotnet publish -c Release -o out-jit

# NativeAOT, self-contained
dotnet publish -r linux-x64 -c Release -
    p:PublishAot=true -o out-aot

# Binary size
ls -lh out-jit/report.dll out-aot/report

# Cold-start latency, best of several runs
for i in 1 2 3 4 5; do /usr/bin/time -v ./out-
    aot/report sample.json; done

```

Representative numbers for a small CLI like the one above (illustrative — measure *your* app; results vary by workload and platform):

Metric	JIT (framework-dependent)	NativeAOT
On-disk size	~0.2 MB app + ~80 MB runtime	~4–8 MB single file
Cold start (hello)	~60–90 ms	~3–10 ms
Working set	~25–35 MB	~8–15 MB
First-request JIT spike	present	none

The startup difference is structural: JIT startup time is spent compiling methods on first call and warming the runtime; AOT pays that at publish time, so the process is doing real work almost immediately. The working-set win comes from no IL meta-data, no JIT scratch, and an aggressively trimmed type system. For a serverless function billed per invocation with cold starts, or a CLI a human waits on, these are not micro-optimizations — they change the product’s feel and cost.

Diagnosing failures: warnings as errors and the analyzer flow

The non-negotiable workflow: **enable the analyzers, treat their warnings as errors, fix at the source.** With

`PublishAot` (or `IsAotCompatible`) the analyzers run during ordinary builds, so

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

means a trim/AOT regression breaks the build at your desk, in PR CI, *before* publish.

When a warning appears, read the IL code:

- `IL2026` — calling a `[RequiresUnreferencedCode]` member. Fix: switch to a trim-safe overload (a `JsonTypeInfo<T>` overload, a generic with `[DynamicallyAccessedMembers]`) or, if reflection is genuinely required, propagate `[RequiresUnreferencedCode]` to your own caller.
- `IL2057` / `IL2070` -family — unanalyzable reflection (a `Type` from a runtime string, or members pulled from an under-annotated `Type` parameter). Fix: flow `[DynamicallyAccessedMembers]` through the call chain, or eliminate the reflection with a source generator.
- `IL3050` — calling a `[RequiresDynamicCode]` member. This is the dangerous one: there is *no* AOT-safe overload for true runtime codegen. Fix: remove `RegexOptions.Compiled`, replace `dynamic` / `Expression.Compile()` / `Reflection.Emit` with static dispatch or generated code.

The forbidden shortcut is `[UnconditionalSuppressMessage]` to silence a warning you didn't actually fix. That is a loaded gun: it compiles, it publishes, and it throws

`MissingMethodException` or returns silently wrong results at runtime, in production, far from the suppression. Suppress *only* when you have proven the pattern is safe (e.g. you preserved the members via `[DynamicDependency]` or the type is rooted), and document why in the message.

Pitfalls to internalize

Library authors discover AOT breakage in someone else's publish step. If you ship a library that reflects over user types and you never set `IsAotCompatible` or `IsTrimmable`, your build is green — the failure surfaces months later in a consumer's `dotnet publish -p:PublishAot=true` as cryptic `IL2026`s pointing into your assembly. Certify libraries with `<IsAotCompatible>true</IsAotCompatible>`, annotate every reflection-using public method with `[RequiresUnreferencedCode]` / `[RequiresDynamicCode]`, and provide source-generator-backed alternatives. The contract belongs in your package, not in your consumer's bug tracker.

Suppressing instead of fixing trades a build warning for a runtime crash. A suppressed `IL2026` is a `MissingMethodException` waiting for the exact input that exercises the trimmed path. The warning was the cheap signal; the production exception is the expensive one. Fix at the source.

“It runs under JIT, so it'll run under AOT” is false. AOT removes capabilities the JIT provides — runtime code generation foremost. Code exercising `dynamic`, `Expression.Compile()`, `Reflection.Emit`, runtime proxy generation, or `MakeGenericType` over value types it never statically instantiated can pass every JIT test and throw on the first AOT run. The only reliable validation is to **publish with `PublishAot=true` and run your full integration suite against the native binary** in CI. Treat the AOT publish as a first-class test target, not an afterthought before release.

When *not* to reach for NativeAOT

It is the wrong tool when your architecture depends on the open world: plugin hosts that load assemblies compiled after your build, ORMs and serializers that reflect over arbitrary user types without source generation, anything built on `dynamic` or expression compilation you can't refactor, or large apps with deep reflection-heavy dependency graphs you don't control. In those cases trimming alone (without full AOT) may still shrink the bi-

nary while keeping the JIT — a softer step. And for long-running services where startup is amortized over days of uptime, the JIT's adaptive, tiered, PGO-driven optimization can produce *faster steady-state throughput* than AOT's static code; the AOT win is startup and footprint, not peak throughput. Choose NativeAOT where cold start, working set, and deployment simplicity dominate — and verify the closed-world assumption holds before you commit.

Sources: - [Native AOT deployment overview — Microsoft Learn](#) - [Optimizing AOT deployments — Microsoft Learn](#) - [Prepare .NET libraries for trimming — Microsoft Learn](#) - [IL3050 warning — Microsoft Learn](#) - [How to use source generation in System.Text.Json — Microsoft Learn](#) - [.NET regular expression source generators — Microsoft Learn](#) - [ASP.NET Core support for Native AOT — Microsoft Learn](#) - [Announcing .NET 10 — .NET Blog](#)

Part VII — Libraries, APIs, and Quality

24 · Modern Collections: Frozen, Immutable, and Span-Friendly

The BCL’s collection story has quietly become one of the richest in any mainstream runtime: read-optimized frozen tables, structurally-shared immutable types, vectorized multi-value search, and zero-copy span windows into the guts of `List<T>` and `Dictionary<TKey, TValue>`. This chapter is about *choosing correctly* under load — which collection matches which access pattern, where the JIT can vectorize for you, and where a convenient abstraction silently allocates or invalidates memory you thought you owned.

The baseline three: List, Dictionary, HashSet

Before reaching for anything exotic, internalize the cost model of the three workhorses, because every “modern” collection is justified relative to them.

`List<T>` is a `T[]` plus a `_size`. Indexing is a bounds-checked array load (the JIT elides the check inside `for (int i = 0; i < list.Count; i++)` when it can prove the range). `Add` is amortized $O(1)$ but doubles capacity on growth — each doubling is a full copy and the old array becomes garbage. If you know the final size, `new List<T>(capacity)` or `EnsureCapacity` removes every intermediate copy and the Geno churn that comes with them.

`Dictionary<TKey, TValue>` uses separate chaining (not open addressing — so no tombstones), backed by two parallel arrays: `_buckets` of `int` indices into `_entries`, and `_entries` of a struct holding `hashCode`, `next`, `key`, `value`. A lookup hashes the key, masks into `_buckets`, then walks a short collision chain through the `next` links in `_entries`. The dominant cost is usually the `EqualityComparer<TKey>.Default` virtual call and the hash itself, not the chain walk. For `string` keys, every lookup hashes the whole string; cache the hash or use a frozen table (below) if the same keys are queried in a tight loop.

`HashSet<T>` shares the dictionary's machinery without values. Use it for membership and set algebra (`UnionWith` , `IntersectWith`), not as a poor man's dictionary.

Two rules carry the rest of the chapter. First: **construction cost amortizes only across reuse** — a structure that is expensive to build but cheap to read wins only when reads vastly outnumber builds. Second: **the comparer is part of the cost** — `StringComparer.Ordinal` is dramatically faster than `OrdinalIgnoreCase` , which is faster than any culture-aware comparer.

A third consideration runs underneath all of them: GC pressure. Each of the three workhorses allocates its backing array(s) on the managed heap, and growth produces garbage — a `List<T>` that doubles five times has produced five dead arrays for the GC to reclaim. For collections of value types, the elements live inline in that array (no per-element boxing), so a `List<int>` of a million entries is one ~4 MB Gen2 array, not a million tiny objects. For reference types the array holds pointers and the GC must trace each live element, so a large `List<string>` costs the collector more to scan than a `List<int>` of the same length. This is why pre-sizing and value-type elements matter beyond raw throughput: they shrink both the allocation count and the tracing surface the GC walks on every collection.

```
// Pre-sizing eliminates intermediate array copies and
// Gen0 garbage.
var ids = new List<int>(capacity: 100_000); // one
// allocation, no growth copies
for (int i = 0; i < 100_000; i++) ids.Add(i); //
// amortized O(1), zero reallocations

// Choosing the comparer is choosing the cost.
var ordinal = new Dictionary<string, int>
    (StringComparer.Ordinal); // fastest
var caseless = new Dictionary<string, int>
    (StringComparer.OrdinalIgnoreCase); // slower
// hash+compare
```

FrozenDictionary and FrozenSet: read-optimized lookups

`FrozenDictionary<TKey,TValue>` and `FrozenSet<T>` (namespace `System.Collections.Frozen`, introduced in **.NET 8**) are immutable collections that spend *extra time at construction* analyzing their keys to pick a specialized, often perfectly-tuned lookup strategy, in exchange for the fastest possible reads. For integer keys they may build a dense or perfect-hash table; for strings they inspect length distributions and substring uniqueness to choose a comparison that touches as few characters as possible. The result is `TryGetValue` / `Contains` that frequently beats `Dictionary` / `HashSet` by 30–50% on read-heavy paths.

The pitch is narrow and you must respect it: **build once (typically at startup), read forever**. There is no `Add`, no `Remove`, no mutation API of any kind — the type is genuinely immutable. Construction is *substantially* more expensive than a `Dictionary` of the same size, because of the analysis pass.

There is a JIT-level reason frozen reads are fast that goes beyond a smarter hash. `FrozenDictionary.ToFrozenDictionary` does not return a single concrete type; it returns one of *many* internal implementations chosen by the analysis —

`Int32FrozenDictionary`, `SmallFrozenDictionary`, `LengthBucketsFrozenDictionary`, `OrdinalStringFrozenDictionary`, and others. Each is sealed and specialized, so when the JIT compiles your `TryGetValue` call against the field's concrete frozen type it can devirtualize and frequently inline the entire lookup, including the comparer. Against a `Dictionary<TKey,TValue>` typed field the comparer call is virtual and the lookup body is not inlined. The frozen win is partly the algorithm and partly the JIT's freedom to specialize a sealed implementation it can see through.

Worked example: freezing a config table for the hot read path

A common shape is configuration loaded once from disk or a database into a mutable dictionary, then queried millions of

times. Freeze it.

```
using System.Collections.Frozen;

// Loaded once at startup from config; never mutated
// afterward.
Dictionary<string, RouteConfig> mutableRoutes =
    LoadRoutesFromDisk();

// ToFrozenDictionary (.NET 8) runs the analysis pass
// and returns an immutable table.
// Pick the comparer deliberately – Ordinal lets the
// frozen builder use the
// cheapest possible string comparison strategy.
FrozenDictionary<string, RouteConfig> routes =

    mutableRoutes.ToFrozenDictionary(StringCompare
        r.Ordinal);

// Hot path, called for every request: a frozen
// lookup, no resize, no mutation locks.
if (routes.TryGetValue(request.Path, out RouteConfig?
    cfg))
    Dispatch(cfg);

readonly record struct RouteConfig(string Handler,
    bool RequireAuth);
```

`FrozenSet<T>` mirrors this with `ToFrozenSet`. Use it for fixed allow-lists, stop-word sets, or feature flags evaluated on every request.

```
// A fixed allow-list checked on every inbound
// message.
FrozenSet<string> allowedOrigins =
    new[] { "clarqo.com", "api.clarqo.com",
        "admin.clarqo.com" }

    .ToFrozenSet(StringComparer.OrdinalIgnoreCase)
    ;

bool ok = allowedOrigins.Contains(request.Origin); //
    vectorized, branch-lean
```

Benchmarking frozen vs. regular

A representative BenchmarkDotNet harness for 10,000 string keys with 100% hit rate looks like this:

```

using BenchmarkDotNet.Attributes;
using System.Collections.Frozen;

[MemoryDiagnoser]
public class FrozenLookupBench
{
    private Dictionary<string, int> _dict = null!;
    private FrozenDictionary<string, int> _frozen =
        null!;
    private string[] _probes = null!;

    [GlobalSetup]
    public void Setup()
    {
        var src = Enumerable.Range(0, 10_000)
            .ToDictionary(i => $"key-{
                i:D5}", i => i);
        _dict = new Dictionary<string, int>(src,
            StringComparer.Ordinal);
        _frozen =
            src.ToFrozenDictionary(StringComparer.Ordinal)
            ;
        _probes = src.Keys.ToArray();
    }

    [Benchmark(Baseline = true)]
    public long Dictionary()
    {
        long sum = 0;
        foreach (var k in _probes) sum += _dict[k];
        return sum;
    }

    [Benchmark]
    public long Frozen()
    {
        long sum = 0;
        foreach (var k in _probes) sum += _frozen[k];
        return sum;
    }
}

// Typical shape of results (your numbers will vary by
// CPU/key distribution):
// | Method      | Mean      | Ratio | Allocated |
// |-----|-----|-----|-----|
// | Dictionary | 78.4 us   | 1.00  | 0 B       |
// | Frozen     | 49.1 us   | 0.63  | 0 B       |

```

Pitfall — rebuilding a `FrozenDictionary` frequently. The construction analysis is the whole reason frozen reads are fast, and it is *expensive*: building a 10k-key frozen dictionary can cost an order of magnitude more than building the equivalent `Dictionary`. If your data changes often — per-request, per-tick, even per-minute under churn — frozen is a net loss; the build cost dwarfs the per-lookup savings. Frozen pays off only when the *ratio* of reads to builds is very high. A table built at startup and read for the process lifetime is the canonical fit; a cache you invalidate every few seconds is an anti-pattern.

The immutable collections family

`System.Collections.Immutable` provides persistent (structurally-shared) collections where every “mutation” returns a new instance and the original is untouched. This is the substrate for thread-safe shared state without locks, and for snapshot semantics. The family includes `ImmutableArray<T>`, `ImmutableList<T>`, `ImmutableDictionary<TKey, TValue>`, `ImmutableHashSet<T>`, `ImmutableSortedDictionary` / `Set`, `ImmutableStack` / `Queue`, plus the builder pattern.

The two you must distinguish are `ImmutableArray<T>` and `ImmutableList<T>`, because choosing wrong has a large, silent cost.

`ImmutableArray<T>` is a `readonly struct` wrapping a plain `T[]`. Reads are array reads — indexing, iteration, and `Span` access are all as fast as a normal array, with zero indirection. The catch: any single-element “mutation” (`Add`, `SetItem`, `RemoveAt`) copies the *entire* array, so it is O(n) per change. It is ideal for collections built once and read many times, exactly where you’d otherwise expose a defensive copy.

`ImmutableList<T>` is an AVL balanced binary tree. Mutations are O(log n) and share most nodes with the prior version — efficient for incremental updates and large collections that change often. But reads pay for it: indexing is O(log n), and **iteration walks tree nodes**, which is far slower and more cache-hostile

than striding an array. Enumerating an `ImmutableList<T>` allocates an enumerator that maintains a stack of tree nodes.

The cache story is the part profilers don't print but hardware counters reveal. An `ImmutableArray<T>` iteration is a linear memory sweep — the prefetcher loves it, and a cold traversal is bounded by sequential bandwidth. An `ImmutableList<T>` traversal chases pointers between tree nodes scattered across the heap; each step risks a cache miss, and the per-element constant factor can be 5–10× the array even though both are $O(n)$. For a few dozen elements nobody notices; for tens of thousands iterated on a request path, picking the tree because it “felt more flexible” is a measurable latency tax.

Pitfall — using `ImmutableList<T>` where

`ImmutableArray<T>` is wanted. If you build a collection once and then iterate it repeatedly (the common case for config, rule sets, route tables), `ImmutableList<T>` makes you pay tree-node overhead on every traversal for a mutation profile you never exercise. Default to `ImmutableArray<T>` for read-mostly immutable data; reach for `ImmutableList<T>` only when you genuinely need cheap incremental edits to a large, frequently-revised collection.

```
using System.Collections.Immutable;

// O(n) per mutation, O(1) read & iteration – right
// for build-once, read-many.
ImmutableArray<string> tags =
    ImmutableArray.Create("ai", "finance",
        "tech");
foreach (var t in tags) { /* array stride, cache-
    friendly */ }

// O(log n) mutation, O(log n) read, slow iteration –
// right only for frequent edits.
ImmutableList<string> editLog =
    ImmutableList<string>.Empty;
editLog = editLog.Add("created").Add("reviewed"); //
    each Add shares most nodes
```

Building large immutable arrays efficiently with the builder

Constructing an `ImmutableArray<T>` element-by-element via `Add` would copy the whole array each time — quadratic. The builder accumulates into a mutable backing array (like a `List<T>`) and hands ownership to an immutable instance at the end, with no final copy when the builder is exactly full.

```
using System.Collections.Immutable;

ImmutableArray<int> BuildSquares(int n)
{
    // Pre-sized builder: one backing array, grows
    // like a List<T> if needed.
    var builder = ImmutableArray.CreateBuilder<int>
        (initialCapacity: n);
    for (int i = 0; i < n; i++)
        builder.Add(i * i);           //
        // amortized O(1), mutable accumulation

    // If Count == Capacity, MoveToImmutable transfers
    // the array with NO copy.
    // ToImmutable() always makes a defensive copy if
    // the builder may be reused.
    return builder.Count == builder.Capacity
        ? builder.MoveToImmutable()    // zero-
        // copy handoff
        : builder.ToImmutable();       // copies
        // the populated prefix
}
```

`MoveToImmutable` requires `Count == Capacity` and nulls out the builder; it is the allocation-optimal path when you know the exact size up front. `ToImmutable` is the safe general path and copies only the populated prefix. For `ImmutableArray`, **never** loop `arr = arr.Add(x)` — that is $O(n^2)$ and a classic profiler hot spot.

CollectionsMarshal: span access into List and Dictionary internals

`CollectionsMarshal` (namespace `System.Runtime.InteropServices`) is the sanctioned, unsafe-

adjacent door into the backing storage of `List<T>` and `Dictionary<TKey, TValue>`. It lets you avoid copies and double lookups — at the cost of taking responsibility for invariants the type would otherwise protect.

`AsSpan(List<T>)` (.NET 5) returns a `Span<T>` over the list's live backing array — `[0, Count)`. You can read and overwrite elements in place with full vectorization and no bounds-check duplication. You cannot change the length through it.

`GetValueRefOrNullRef` and `GetValueRefOrAddDefault` (.NET 6) return a `ref` directly into a dictionary's entry array, eliminating the classic “lookup, then index again to store” double hash.

Worked example: updating a struct value without a double lookup

The textbook anti-pattern with struct-valued dictionaries is `dict[k] = Mutate(dict[k])` — two hashes, two lookups, and a full struct copy in and out. `GetValueRefOrAddDefault` collapses it to a single lookup that returns a writable reference to the value *in place*.


```

using System.Runtime.InteropServices;

struct Counter { public int Hits; public long Bytes; }

void Record(Dictionary<string, Counter> stats, string
    key, int bytes)
{
    // One hash, one probe. 'exists' is false when a
    // default entry was just added.
    ref Counter c = ref
        CollectionsMarshal.GetValueRefOrAddDefault(stats,
            key, out bool exists);

    // 'c' aliases the value slot inside the
    // dictionary's entry array.
    // Mutating fields here writes straight into the
    // dictionary – no copy-back.
    c.Hits++;
    c.Bytes += bytes;

    if (!exists) { /* first sighting of this key, if
        you need to react */ }
}

```

This is the single most valuable `CollectionsMarshal` method for hot aggregation loops (metrics, histograms, word counts). With a non-struct (reference-type) value the `ref` still saves the second lookup, though you typically don't need the in-place mutation.

Pitfall — mutating a `List<T>` while holding a span from `CollectionsMarshal.AsSpan`. The span points at the *current* backing array. Any operation that can resize the list — `Add`, `Insert`, `EnsureCapacity`, `RemoveAt` that shifts, even `Clear` on some paths — may allocate a new array and copy, leaving your span pointing at the old, now-stale array (or worse, writing where it has no effect). The runtime does not invalidate the span for you; there is no `InvalidOperationException` like the enumerator version check. Treat the span as valid only for the duration of a tight read/overwrite loop with no structural changes:

```
var list = new List<int>(capacity: 4) { 1, 2, 3, 4 };
Span<int> span = CollectionsMarshal.AsSpan(list);

list.Add(5);           // capacity exceeded -> new
                       // array allocated, list copied
span[0] = 999;         // writes into the OLD array;
                       // list[0] is still 1. Silent bug.
```

The same caution applies to `GetValueRefOrNullRef`: the `ref` is invalidated by any dictionary insertion that triggers a resize/rehash. Read or write through it immediately; never hold it across an `Add`.

Searching and sorting spans

`MemoryExtensions` gives spans a full search/sort surface that the JIT vectorizes aggressively: `BinarySearch`, `Sort`, `IndexOf`, `Contains`, `IndexOfAny`, and the `*Except` variants.

`Span<T>.BinarySearch` requires the span to be sorted and returns the index of a match, or the bitwise complement (`~insertionPoint`) of where the element would go — the same convention as `Array.BinarySearch`.

```
ReadOnlySpan<int> sorted = stackalloc int[] { 2, 5, 8,
                                             13, 21 };
int idx = sorted.BinarySearch(13);           // 3
int miss = sorted.BinarySearch(10);          // ~3 =>
                                             // insertion point is 3
if (miss < 0) idx = ~miss;                    // recover
                                             // the would-be position
```

SearchValues: vectorized multi-value search

`SearchValues<T>` (.NET 8) precomputes an optimal representation of a *set of needle values* so that

`IndexOfAny` / `ContainsAny` over a span runs as a single vectorized pass instead of an $O(\text{needle} \times \text{haystack})$ double loop. The set is analyzed once at construction; the search uses SIMD (e.g. SSE/AVX or ARM NEON) where the values' shape allows. .NET 9 extended `SearchValues<string>` so you can search a

span for the first occurrence of any of several *substrings*, with a `StringComparison`.

The cardinal rule (enforced by analyzer **CA1870**): **create the `SearchValues<T>` once in a `static readonly` field and reuse it**. Recreating it per call throws away the entire benefit.

```
using System.Buffers;

static class Sanitizer
{
    // Built once; reused for the process lifetime.
    // CA1870 will flag inline creation.
    private static readonly SearchValues<char>
        s_unsafe =
            SearchValues.Create("<&\"'\\0");

    // Vectorized scan: returns -1 fast if the span is
    // clean, common-case win.
    public static bool
        ContainsUnsafe(ReadOnlySpan<char> text) =>
            text.IndexOfAny(s_unsafe) >= 0;

    public static int FirstUnsafe(ReadOnlySpan<char>
        text) =>
            text.IndexOfAny(s_unsafe);
}

// .NET 9: multi-substring search over a char span.
static class TokenScan
{
    private static readonly SearchValues<string>
        s_keywords =
            SearchValues.Create(["SELECT", "INSERT",
                "UPDATE", "DELETE"],
                StringComparison.OrdinalIgnoreCase);

    public static int FirstKeyword(ReadOnlySpan<char>
        sql) =>
        sql.IndexOfAny(s_keywords); // first index
        where any keyword begins
}
```

For ASCII byte scanning (parsers, protocol framing), `SearchValues<byte>` is the same idea over `ReadOnlySpan<byte>` and is the backbone of high-performance JSON/HTTP parsing in the BCL itself.

Why is this faster than a hand-rolled `for` loop with a `switch`? A scalar loop tests one element per iteration with a chain of comparisons and unpredictable branches — the branch predictor mispredicts on irregular data and stalls the pipeline.

`SearchValues` selects a *branch-free vectorized* algorithm at construction based on the value set: for a small contiguous range it may use a bitmap-and-shuffle (`PSHUFF / TBL`); for a handful of scattered values it broadcasts each needle across a SIMD register and ORs the comparison masks, testing 16–32 bytes per instruction. The “found nothing” path — overwhelmingly the common case in validation — collapses to a few SIMD compares over the whole span with no per-element branching at all. That is the difference between memory-bandwidth-bound and branch-mis-prediction-bound, and it is why the analyzer is insistent about caching the instance: rebuilding the vectorized representation per call would erase the win it exists to provide.

PriorityQueue and other newer BCL collections

`PriorityQueue<TElement, TPriority>` (.NET 6) is an array-backed quaternary (4-ary) min-heap. `Enqueue` and `Dequeue` are $O(\log n)$; `Peek` is $O(1)$. It is **not stable** — equal priorities dequeue in an unspecified order — and it is **not enumerable in priority order** (`UnorderedItems` is heap-layout order). Supply a custom `IComparer<TPriority>` to invert to a max-heap or to order composite priorities.

Worked example: a priority scheduler, contrasted with a sorted list

```
readonly record struct Job(string Name);

void RunScheduler()
{
    // Lower priority value = runs sooner (min-heap default).
    var pq = new PriorityQueue<Job, int>();
    pq.Enqueue(new Job("backfill"), priority: 10);
    pq.Enqueue(new Job("user-click"), priority: 1);
    pq.Enqueue(new Job("email-flush"), priority: 5);

    // EnqueueDequeue / DequeueEnqueue avoid a
    // separate heapify when swapping.
    while (pq.TryDequeue(out Job job, out int prio))
        Console.WriteLine($"[{prio}] {job.Name}");
    // [1] user-click
    // [5] email-flush
    // [10] backfill
}
```

The naive alternative — keep a `List<(int prio, Job)>` and `Sort()` (or `OrderBy`) on every insert or before each pop — is $O(n \log n)$ *per operation* and allocates iterators with LINQ. For a steady stream of enqueues and dequeues the heap is asymptotically and practically superior:

```
// Anti-pattern for streaming workloads: re-sorts the
// whole list on every pop.
var bag = new List<(int prio, Job job)>();
bag.Add((10, new Job("backfill")));
bag.Add((1, new Job("user-click")));
bag.Sort((a, b) => a.prio.CompareTo(b.prio)); // O(n
// log n) EVERY time you mutate
var next = bag[0]; bag.RemoveAt(0);           //
// RemoveAt(0) is O(n) too
```

Use the sorted-list shape only when you also need ordered *iteration* of the whole set or random access by rank; for “give me the most urgent item” semantics, `PriorityQueue` wins. When you need a sorted collection that stays sorted and supports range queries, prefer

`SortedSet<T>` / `SortedDictionary<TKey,TValue>` (red-black trees, $O(\log n)$ ops, ordered iteration).

OrderedDictionary<TKey,TValue>

The generic `OrderedDictionary<TKey,TValue>` (.NET 9, in `System.Collections.Generic` — distinct from the legacy non-generic `System.Collections.Specialized` one) gives you $O(1)$ keyed lookup *and* a stable insertion order with index access. It exposes `GetAt(index)`, `SetAt`, `Insert(index, key, value)`, `RemoveAt`, and `IndexOf(key)`. Use it for LRU-ish structures, ordered form fields, or anywhere you previously paired a `Dictionary` with a parallel `List<TKey>`.

```
var od = new OrderedDictionary<string, int>();
od.Add("first", 1);
od.Add("second", 2);
od.Insert(0, "zeroth", 0);           // now at position
                                     // 0; others shift right
// GetAt returns KeyValuePair<TKey,TValue>;
//   deconstructed here into (k, v).
(string k, int v) = od.GetAt(1);     // ("first", 1) -
                                     // position 1 after the insert
int pos = od.IndexOf("second");      // 2
```

Collection expressions and CollectionBuilder

Collection expressions (**C# 12**) — the `[a, b, ..spread]` syntax — are the uniform way to build collections, and crucially they let the compiler pick an *efficient* construction strategy per target type. For `Span<T>` / `ReadOnlySpan<T>` targets the compiler may `stackalloc`; for arrays it emits a direct fill; for `List<T>` it pre-sizes. The spread operator `..` inlines another sequence.

```
int[] a = [1, 2, 3];
List<int> b = [0, ..a, 4];           // [0,1,2,3,4], pre-sized, no LINQ
ReadOnlySpan<int> c = [10, 20, 30]; // may be stack-allocated, zero heap
ImmutableArray<int> d = [1, 1, 2, 3, 5]; // builds via the type's CollectionBuilder
```

That last line works because `ImmutableArray<T>` is annotated with `[CollectionBuilder]`, telling the compiler which factory turns a span of elements into the type. You can opt your own types in with the same attribute (**C# 12**): point it at a static method taking `ReadOnlySpan<T>`.

```
using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

[CollectionBuilder(typeof(RingBuilder),
    nameof(RingBuilder.Create))]
public readonly struct Ring<T> : IEnumerable<T>
{
    private readonly T[] _items;
    public Ring(ReadOnlySpan<T> items) => _items =
        items.ToArray();
    public int Count => _items.Length;
    public ReadOnlySpan<T> AsSpan() => _items;

    // A [CollectionBuilder] target MUST expose an
    // iteration type – either a public
    // GetEnumerator() or IEnumerable<T>. Without it
    // the element type is unknown and
    // 'Ring<int> r = [...]' does not compile.
    public IEnumerator<T> GetEnumerator() =>
        ((IEnumerable<T>)_items).GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() =>
        GetEnumerator();
}

public static class RingBuilder
{
    // The compiler calls this for a collection
    // expression targeting Ring<T>.
    // The signature must be static and take exactly
    // one ReadOnlySpan<T> parameter.
    public static Ring<T> Create<T>(ReadOnlySpan<T>
        items) => new(items);
}

Ring<int> r = [1, 2, 3, 4]; // lowered to
    RingBuilder.Create(span over inline storage)
```

A genuinely current detail: **C# 14** / **.NET 10** adds *first-class implicit* `Span<T>` / `ReadOnlySpan<T>` conversions, so a collection expression — and ordinary code — can flow an array or list into span-typed parameters and locals without an explicit

`.AsSpan()` in many more positions. Combined with **C# 13** `params` collections (`params ReadOnlySpan<T>`), a vararg-style API can now accept `[1, 2, 3]` and receive a stack-allocated span with no array allocation at all:

```
// C# 13: params ReadOnlySpan<T>. Call site allocates  
nothing on the heap.  
static int Sum(params ReadOnlySpan<int> xs)  
{  
    int s = 0;  
    foreach (var x in xs) s += x;  
    return s;  
}  
  
int total = Sum(1, 2, 3, 4); // stack-allocated  
span; no int[] on the heap
```

Exposing safe views: `ReadOnly` wrappers

When a type holds a mutable collection internally but must expose it, returning the live collection invites external mutation. The cheap, allocation-light options:

- `ReadOnlyList<T>` / `ReadOnlyDictionary<TKey, TValue>` as the *return type* documents intent but does **not** prevent a caller from casting back to the concrete mutable type. It is a contract, not a guarantee.
- `ReadOnlyCollection<T>` / `ReadOnlyDictionary<TKey, TValue>` wrap the live collection: mutation attempts throw, and there's no back-cast to the mutable type. The wrapper is a thin object that forwards reads, so it sees later mutations to the underlying collection (it's a *view*, not a snapshot).
- `list.AsReadOnly()` is the idiomatic factory; **.NET 8** broadened the surface with `CollectionExtensions.AsReadOnly` overloads for `IList<T>` ($\rightarrow$ `ReadOnlyCollection<T>`) and `IDictionary<TKey, TValue>` ($\rightarrow$ `ReadOnlyDictionary<, >`), and **.NET 9** added the `ISet<T>` overload returning the new `ReadOnlySet<T>`.

`Array.AsReadOnly<T>` itself is much older (Framework 2.0).

- For a true *immutable snapshot* that won't reflect later changes, copy into `ImmutableArray<T>` or `FrozenSet<T>`.

```
public sealed class Catalog
{
    private readonly List<string> _skus = [];

    // View: forwards reads, blocks writes, reflects
    // later Add()s to _skus.
    public IReadOnlyList<string> Skus =>
        _skus.AsReadOnly();

    // Snapshot: a frozen copy, decoupled from future
    // mutations of _skus.
    public ImmutableArray<string> SnapshotSkus() =>
        [.._skus];
}
```

Choose a *view* when callers should observe live state and a *snapshot* when they must not — the distinction matters under concurrency, where a view handed to another thread can change under it while an `ImmutableArray<T>` cannot.

Tying it together: a lookup-heavy request classifier

This worked example combines the chapter's ideas into a realistic hot path: classify inbound requests using a frozen route table, a reused `SearchValues` to reject malformed paths, and in-place struct aggregation for per-route metrics.

```

using System.Buffers;
using System.Collections.Frozen;
using System.Runtime.InteropServices;

public sealed class RequestClassifier
{
    // Built once at startup from config – read on
    // every request thereafter.
    private readonly FrozenDictionary<string,
        RouteConfig> _routes;

    // Reused vectorized scan for characters never
    // valid in a normalized path.
    private static readonly SearchValues<char>
        s_illegal =
            SearchValues.Create(" \t\r\n<>\"\\0");

    // Per-route metrics, mutated in place via
    // CollectionsMarshal.
    private readonly Dictionary<string, RouteStats>
        _stats = new(StringComparer.Ordinal);

    public
        RequestClassifier(ReadOnlyDictionary<string,
            RouteConfig> config) =>
        // Ordinal comparer matters twice: the frozen
        // builder picks the cheapest
        // string strategy, AND StringComparer.Ordinal
        // implements
        // IAlternateEqualityComparer<ReadOnlySpan<char>,
        // string>, which is what
        // makes the span-keyed alternate lookup below
        // legal.
        _routes =
            config.ToFrozenDictionary(StringComparer.Ordinal);

    public Verdict Classify(ReadOnlySpan<char> path,
        int byteLen)
    {
        if (path.IndexOfAny(s_illegal) >= 0) //
            SIMD reject, common fast path
            return Verdict.Malformed;

        // .NET 9:
        FrozenDictionary.GetAlternateLookup<ReadOnlySpan<char>>() probes
        // the frozen table directly from the span –
        // no string materialized on the
        // hot path. Requires the Ordinal comparer
        // (set in the constructor).
    }

```

```

var routeLookup =
    _routes.GetAlternateLookup<ReadOnlySpan<char>>
        ();
if (!routeLookup.TryGetValue(path, out
    RouteConfig cfg))
    return Verdict.NotFound;

// Metrics also stay allocation-free: a span-
keyed alternate lookup over
// the mutable stats dictionary, fed to the
.NET 9 CollectionsMarshal
// overload that takes a
Dictionary<,>.AlternateLookup<TAlternateKey>
and
// returns a writable ref to the value slot –
one hash+probe, in place.
var statsLookup =
    _stats.GetAlternateLookup<ReadOnlySpan<char>>
        ();
ref RouteStats st = ref CollectionsMarshal
    .GetValueRefOrAddDefault(statsLookup,
        path, out _);
st.Hits++;
st.TotalBytes += byteLen;

return cfg.RequireAuth ? Verdict.NeedsAuth :
    Verdict.Allow;
}
}

public readonly record struct RouteConfig(string
    Handler, bool RequireAuth);
public struct RouteStats { public long Hits; public
    long TotalBytes; }
public enum Verdict { Allow, NeedsAuth, NotFound,
    Malformed }

```

Every choice here is load-bearing, and the whole path is allocation-free for both the reject and the matched cases. The route table is frozen because it's built once and read per request; `s_illegal` is `static readonly` so the SIMD tables are computed exactly once; the illegal-character check runs first so malformed input is rejected by a single vectorized scan. The keyed lookups never materialize a string: **.NET 9's** `AlternateLookup` feature exposes a `ReadOnlySpan<char>`-keyed view over both the `FrozenDictionary` (`GetAlternateLookup<ReadOnlySpan<char>>()`) and the mutable stats `Dictionary` (fed to the new

`CollectionsMarshal.GetValueRefOrAddDefault` overload that accepts a `Dictionary<,>.AlternateLookup<TAlternateKey>`). Both work only because the dictionaries use `StringComparer.Ordinal` , which implements `IEqualityComparer<ReadOnlySpan<char>, string>` ; a culture-aware comparer would throw at the `GetAlternateLookup` call. This is the payoff of the chapter's throughline — frozen's read speed *and* zero-allocation span probing, with no `ToString()` anywhere on the hot path.

Decision summary

- **Read-mostly, fixed keys, built at startup** → `FrozenDictionary` / `FrozenSet` . Never rebuild on a hot cadence.
- **Read/write keyed, general purpose** → `Dictionary` / `HashSet` ; reach for `AlternateLookup` to probe by span without allocating keys.
- **Immutable, build-once read-many** → `ImmutableArray<T>` (array-speed reads). Build it with a pre-sized builder and `MoveToImmutable` .
- **Immutable with frequent incremental edits** → `ImmutableList<T>` / `ImmutableDictionary` — but only then; you pay tree overhead on every read.
- **In-place aggregation / avoiding double lookups** → `CollectionsMarshal.GetValueRefOrAddDefault` ; never hold the `ref` /span across a structural mutation.
- **Multi-value/substring search over spans** → cached `SearchValues<T>` .
- **“Most urgent first” streaming** → `PriorityQueue<TElement,TPriority>` ; for ordered iteration use `SortedSet` / `SortedDictionary` ; for keyed + insertion-ordered use the generic `OrderedDictionary<TKey,TValue>` .
- **Exposing internal state** → `AsReadOnly()` for a live view, `[..collection]` into `ImmutableArray<T>` for a snapshot.

The throughline: modern .NET collections push *cost to the moment of construction* (frozen analysis, `SearchValues` precompute, immutable builders) so the steady-state read path is as close to raw memory as the type system allows. Pay that cost once, in the right place, and read for free forever.

25 · System.Text.Json in Depth

`System.Text.Json` is the JSON stack that ships in the box with .NET, and by .NET 10 it has matured from “the fast minimal serializer” into a layered toolkit that spans a reflection-driven convenience API, a compile-time source generator built for NativeAOT and trimming, a read-only and a mutable DOM, and a hand-tuned span layer for zero-allocation parsing. This chapter walks the whole stack from the top — `JsonSerializer` and `JsonSerializerOptions` — down to `Utf8JsonReader` / `Utf8JsonWriter`, with an emphasis on the tradeoffs an expert actually makes: when reflection mode quietly breaks under AOT, why a misused `JsonSerializerOptions` costs you a metadata rebuild on every call, and where the source generator earns its keep. Every API here is real BCL surface as of .NET 10; version introductions are flagged in prose.

The convenience layer: `JsonSerializer` and `JsonSerializerOptions`

The entry point everyone meets first is the static `JsonSerializer`. The synchronous string-based overloads are fine for small payloads and tests; the `Utf8` and `Stream` overloads avoid a UTF-16 round trip and are what you should reach for in servers.

```

using System.Text.Json;

public sealed record Product(int Id, string Name,
    decimal Price, bool InStock);

// String round-trip – convenient, but transcodes UTF-
// 8 <=> UTF-16.
string json = JsonSerializer.Serialize(new Product(7,
    "Cable", 12.5m, true));

// UTF-8 byte path – no transcoding, the serializer's
// native encoding.
byte[] utf8 = JsonSerializer.SerializeToUtf8Bytes(new
    Product(7, "Cable", 12.5m, true));
Product? p = JsonSerializer.Deserialize<Product>
    (utf8);

```

`System.Text.Json` speaks UTF-8 natively. `Serialize<T>(value)` returns a `string`, which means it serializes to UTF-8 internally and then transcodes to UTF-16 for you.

`SerializeToUtf8Bytes` and the `Utf8JsonWriter`-based overloads skip that step. In a high-throughput pipe — an ASP.NET Core endpoint, a message handler — prefer the byte/stream paths.

Behavior is governed by `JsonSerializerOptions`. The defaults are deliberately strict and minimal: **PascalCase property names, case-sensitive matching, no indentation, integer enums**. Two facts surprise people coming from `Newtonsoft.Json`:

1. **Property-name matching is case-sensitive by default.** `{"name":"x"}` will *not* bind to a `Name` property unless you opt into case-insensitivity.
2. **The default emits PascalCase**, because it echoes your CLR member names verbatim. Web APIs almost always want camelCase.

The fix for both is the web preset.

`JsonSerializerDefaults.Web` flips `PropertyNamingPolicy` to camelCase, sets `PropertyNameCaseInsensitive = true`, and reads numbers from JSON strings. ASP.NET Core's minimal APIs and MVC already use these defaults, which is why “it works

in my controller but not in my console app” is a perennial confusion.

```
// Pre-built web preset. Construct once, reuse forever
// (see caching note below).
var web = new JsonSerializerOptions(JsonSerializerDefaults.Web);

// .NET 9 added a shared, frozen singleton of exactly
// this preset:
JsonSerializerOptions cached =
    JsonSerializerOptions.Web;    // read-only,
    never mutate
JsonSerializerOptions empty =
    JsonSerializerOptions.Default; // PascalCase,
    frozen
```

`JsonSerializerOptions.Web` (introduced in **.NET 9**) is a pre-frozen singleton equivalent to `new JsonSerializerOptions(JsonSerializerDefaults.Web)` — use it instead of allocating your own copy when you don’t need further customization. `JsonSerializerOptions.Default` (**.NET 8**) is the frozen PascalCase default.

Naming policies, casing, and enums

Naming policy controls how CLR member names map to JSON keys. The built-ins:

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    // since 3.0: modelName
    DictionaryKeyPolicy =
        JsonNamingPolicy.SnakeCaseLower, // .NET 9:
        model_number
    WriteIndented = true,
    // .NET 9: customize the writer's indentation
    // glyph and width.
    IndentCharacter = '\t',
    IndentSize = 1,
};
```

`JsonNamingPolicy.CamelCase` has been there since the beginning. **.NET 9** added four more: `SnakeCaseLower`,

`SnakeCaseUpper`, `KebabCaseLower`, and `KebabCaseUpper`.

Before .NET 9 you hand-rolled a `JsonNamingPolicy` subclass for `snake_case`; now it's a one-liner. Note that

`PropertyNamingPolicy` and `DictionaryKeyPolicy` are independent — the former renames declared members, the latter renames `Dictionary<string, T>` keys.

Enums serialize as their numeric value by default, which is compact but opaque and brittle across renumbering. To emit strings, attach `JsonStringEnumConverter`. The non-generic

`JsonStringEnumConverter` relies on runtime reflection over the enum and is **not** AOT-safe; .NET 8 introduced the generic `JsonStringEnumConverter<TEnum>`, which is the only string-enum converter the NativeAOT runtime supports.

```
using System.Text.Json.Serialization;

public enum Severity { Low, Medium, High }

public sealed class Alert
{
    // AOT-safe, statically analyzable – prefer this
    // over
    [JsonConverter(typeof(JsonStringEnumConverter))]
    public Severity Level { get; set; }
}
```

For source-gen contexts you can globally request string enums without per-property attributes via

`JsonSourceGenerationOptions.UseStringEnumConverter = true` (.NET 9) — shown in the source-gen section.

The cardinal performance rule: cache your options

`JsonSerializerOptions` looks like a humble bag of flags, but the **first** serialize/deserialize with a given instance triggers construction of a per-type metadata cache (contract resolution, converter selection, property ordering) keyed off that exact instance. After that first call the instance is implicitly frozen and the cache

is reused. If you `new JsonSerializerOptions{...}` on every request, you throw that cache away every time and pay full reflection/contract cost on each call — easily a 10–100× regression on hot paths.

```
public static class JsonConfig
{
    // Static, constructed once, frozen on first use.
    // THIS is the pattern.
    public static readonly JsonSerializerOptions
        Default = new(JsonSerializerDefaults.Web)
    {
        Converters = { new
            JsonSerializerEnumConverter<Severity>() },
        DefaultIgnoreCondition =
            JsonIgnoreCondition.WhenWritingNull,
    };
}

// Hot path – reuses the cached metadata.
string s = JsonSerializer.Serialize(alert,
    JsonConfig.Default);
```

`MakeReadOnly()` (.NET 8) freezes an instance explicitly without performing a serialization, which is handy when you want to fail fast on accidental late mutation or pre-warm at startup. Once frozen, any mutation throws `InvalidOperationException`. The parameterless overload **throws** `InvalidOperationException` (“must specify a `TypeInfoResolver` setting before being marked as read-only”) if no resolver is configured; use the `MakeReadOnly(populateMissingResolver: true)` overload to have it inject a reflection-based `DefaultJsonTypeInfoResolver` for you (which reintroduces trim warnings) — relevant to the source-gen discussion that follows. .NET 10 added `JsonSerializerOptions.Strict`, a ready-made preset bundling stricter validation (e.g. duplicate-property rejection, throwing on unmapped members) for security-sensitive input.

Source-generated serialization

Everything above defaults to **reflection mode**: at runtime the serializer reflects over your types to discover properties, build getters/setters, and select converters. It’s flexible and zero-cere-

mony, but it has two serious problems in modern deployment targets:

- **Trimming/NativeAOT.** The trimmer can't statically see which members reflection will touch, so it either keeps everything (defeating trimming) or trims members the serializer needs at runtime, producing `NotSupportedException` or silently-empty objects. Reflection-mode `System.Text.Json` is fundamentally **trim-unsafe and AOT-incompatible**; the build emits IL2026/IL3050 warnings.
- **Startup cost.** Reflecting over a large type graph and emitting dynamic IL on first use adds measurable cold-start latency.

The **source generator** (`JsonSerializerContext`, introduced in **.NET 6**) solves both. You declare a `partial` context class, annotate it with `[JsonSerializable(typeof(T))]` for each root type, and at compile time the generator emits strongly-typed `JsonTypeInfo<T>` metadata and (optionally) fast-path serialization code. No runtime reflection, no dynamic IL, fully trim/AOT-safe, faster startup.

```

using System.Text.Json;
using System.Text.Json.Serialization;

public sealed record Order(int Id, string Customer,
    decimal Total, OrderLine[] Lines);
public sealed record OrderLine(string Sku, int Qty,
    decimal UnitPrice);

[JsonSourceGenerationOptions(
    PropertyNamingPolicy =
        JsonKnownNamingPolicy.CamelCase,
    DefaultIgnoreCondition =
        JsonIgnoreCondition.WhenWritingNull,
    UseStringEnumConverter = true)] //
    .NET 9: global string enums
[JsonSerializable(typeof(Order))]
[JsonSerializable(typeof(OrderLine))] //
    optional; reachable types are discovered
public partial class OrderJsonContext :
    JsonSerializerContext
{
}

```

Usage uses the generated, per-type `JsonTypeInfo<T>` accessors — there is an overload of every `JsonSerializer` method that takes them, and they're what make the whole call statically analyzable:

```

var order = new Order(101, "Acme", 59.97m,
    [new OrderLine("CBL-1", 3, 19.99m)]);

// Serialize with the generated metadata – no
// reflection, AOT-safe.
string json = JsonSerializer.Serialize(order,
    OrderJsonContext.Default.Order);

// Deserialize the same way.
Order? back = JsonSerializer.Deserialize(json,
    OrderJsonContext.Default.Order);

// You can also hang the context off an options
// instance:
var options = new JsonSerializerOptions {
    TypeInfoResolver = OrderJsonContext.Default };
string json2 = JsonSerializer.Serialize(order,
    options);

```

Metadata mode vs serialization-optimization mode

The generator emits in one of two modes, set per-context via `[JsonSourceGenerationOptions(GenerationMode = ...)]` or per-type via `[JsonSerializable(..., GenerationMode = ...)]`:

- `JsonSourceGenerationMode.Metadata` emits `JsonTypeInfo<T>` metadata that drives the *general-purpose* (de)serialization engine. It supports **both serialization and deserialization** and the full feature set — custom converters, reference handling, every option. This is what you want for deserialization and for anything using advanced options.
- `JsonSourceGenerationMode.Serialization` emits a hand-rolled fast-path writer that calls `Utf8JsonWriter` directly with no metadata indirection. It is **serialize-only**, doesn't support deserialization, and only applies when the configured options are compatible with the fast path (it silently falls back to the metadata engine otherwise). This is the fastest serialization mode available.

If you specify neither, the generator emits **both** (`Metadata` | `Serialization`), giving you fast-path writes plus full deserialization — at the cost of more generated code. For an API that only ever *writes* a type, `Serialization` mode minimizes output and maximizes throughput; for a type you both read and write, take the default. Mark each `[JsonSerializable]` appropriately:

```
[JsonSerializable(typeof(Order),
    GenerationMode =
        JsonSourceGenerationMode.Serialization)] //
    write-only fast path
[JsonSerializable(typeof(InboundCommand),
    GenerationMode =
        JsonSourceGenerationMode.Metadata)]      //
    needs deserialize
public partial class ApiJsonContext :
    JsonSerializerContext { }
```

When not to use source-gen: if you're not trimming or running AOT, your type graph is highly dynamic (decided at run-time, `object`-typed, or driven by third-party types you can't annotate), or you lean on reflection-mode-only features, the generator buys you little and costs you maintenance. The honest default for a JIT-compiled server is: use reflection mode unless you're going AOT or have a measured startup/throughput problem. The honest default for a **library, a container image you're trimming, or any NativeAOT target** is: source-gen from day one, because reflection mode will not survive the trimmer.

You can compose multiple resolvers via

`TypeInfoResolverChain` (.NET 8), letting you stack a source-gen context and a reflection fallback (`DefaultJsonTypeInfoResolver`) — though mixing in the reflection resolver reintroduces trim warnings.

Custom converters: `JsonConverter<T>`

When the built-in contract for a type is wrong — a value type with a non-default wire format, a domain primitive, a legacy date encoding — write a `JsonConverter<T>`. The contract is two methods, `Read` and `Write`, operating directly on the span-level reader/writer. Here is a converter for a value type representing money as a decimal-string on the wire (common in finance APIs that refuse to trust IEEE floats):

```

using System.Text.Json;
using System.Text.Json.Serialization;
using System.Globalization;

public readonly record struct Money(decimal Amount,
    string Currency)
{
    // Fixed 2-dp wire form so 12.5m round-trips as
    // "12.50", not "12.5".
    public override string ToString() => $"
        {Amount.ToString("F2",
            CultureInfo.InvariantCulture)} {Currency}";
}

public sealed class MoneyConverter :
    JsonSerializer<Money>
{
    public override Money Read(ref Utf8JsonReader
        reader, Type typeToConvert,
        JsonSerializerOptions options)
    {
        if (reader.TokenType != JsonTokenType.String)
            throw new JsonException($"Expected string
                for Money, got {reader.TokenType}.");

        string raw = reader.GetString(!);
        // e.g. "12.50 USD"
        int space = raw.IndexOf(' ');
        if (space <= 0 || space == raw.Length - 1)
            throw new JsonException($"Malformed Money
                value '{raw}'.");

        decimal amount = decimal.Parse(
            raw.AsSpan(0, space), NumberStyles.Number,
            CultureInfo.InvariantCulture);
        string currency = raw[(space + 1)..];
        return new Money(amount, currency);
    }

    public override void Write(Utf8JsonWriter writer,
        Money value, JsonSerializerOptions options)
        => writer.WriteStringValue(value.ToString());
}

```

Register it on options (cache the options!) or per-member with
`[JsonConverter(typeof(MoneyConverter))]` :

```
var opts = new JsonSerializerOptions { Converters = {
    new MoneyConverter() } };
string j = JsonSerializer.Serialize(new Money(12.5m,
    "USD"), opts); // "12.50 USD"
Money m = JsonSerializer.Deserialize<Money>("\"12.50
    USD\"", opts);
```

Several converter-authoring rules are non-obvious and worth committing to memory:

- **Read** must leave the reader positioned on the last token of your value, not past it. For a scalar like a string you read exactly one token and stop. If you read a JSON *object* or *array*, you must consume through its matching end token; leaving the cursor misaligned corrupts the parse for the next property.
- **Utf8JsonReader** is a **ref struct**. You receive it **ref**. You cannot capture it into a lambda, a field, an **async** state machine, or an iterator. Do all parsing synchronously inside **Read**.
- **For NativeAOT, prefer **JsonConverter<T>** over **JsonConverterFactory**** when possible; factories that construct converters reflectively (**Activator.CreateInstance**, generic **MakeGenericType**) reintroduce trim/AOT hazards. A factory for **Money** -like closed types is fine; a factory that builds converters for *open* generics at runtime is the risky pattern.
- **Converters are discovered per-options and cached**, so registering one is cheap after the first call — another reason to reuse the options instance.

The span hot path: **Utf8JsonReader** and

Utf8JsonWriter

Beneath **JsonSerializer** sit two low-level, allocation-free, forward-only primitives. **Utf8JsonReader** is a **ref struct** cursor over a **ReadOnlySpan<byte>** (or **ReadOnlySequence<byte>**); **Utf8JsonWriter** writes tokens to an **IBufferWriter<byte>** or **Stream**. They are the engine the serializer is built on, and you

drop to them when you need to parse or emit JSON with **zero per-token allocation** and total control — telemetry ingestion, log shipping, protocol bridges, hot inner loops where even the serializer's overhead shows up in a profile.

A manual reader loop is a state machine over `reader.Read()`, which advances one token at a time and returns `false` at the end of the buffer:

```

using System.Text.Json;

// Parse {"id":7,"name":"Cable","tags":
// ["a","b"],"price":12.5} without allocating a
// POJO.
public static (int Id, string? Name, decimal Price,
    int TagCount) ParseProduct(ReadOnlySpan<byte>
    utf8)
{
    var reader = new Utf8JsonReader(utf8,
        isFinalBlock: true, state: default);

    int id = 0, tagCount = 0;
    string? name = null;
    decimal price = 0m;

    // Advance to the opening object brace.
    if (!reader.Read() || reader.TokenType !=
        JsonTokenType.StartObject)
        throw new JsonException("Expected JSON
            object.");

    while (reader.Read())
    {
        if (reader.TokenType ==
            JsonTokenType.EndObject)
            break;

        // We're on a property name; ValueTextEquals
        // compares against UTF-8 without allocating.
        if (reader.TokenType !=
            JsonTokenType.PropertyName)
            throw new JsonException($"Unexpected token
                {reader.TokenType}.");

        if (reader.ValueTextEquals("id"u8))
        {
            reader.Read();
            id = reader.GetInt32();
        }
        else if (reader.ValueTextEquals("name"u8))
        {
            reader.Read();
            name = reader.GetString(); //
            // allocates the string; intrinsic to the result
        }
        else if (reader.ValueTextEquals("price"u8))
        {
            reader.Read();
            price = reader.GetDecimal();
        }
    }
}

```

```

        else if (reader.ValueTextEquals("tags"u8))
        {
            reader.Read(); //
            StartArray
            if (reader.TokenType !=
                JsonTokenType.StartArray)
                throw new JsonException("Expected
                    array for 'tags'.");
            while (reader.Read() && reader.TokenType
                != JsonTokenType.EndArray)
                tagCount++; //
            count elements, never materialize them
        }
        else
        {
            reader.Skip(); //
            skip unknown property's entire value subtree
        }
    }

    return (id, name, price, tagCount);
}

```

The allocation story here is the point:

`ValueTextEquals("id"u8)` compares the property name against a **UTF-8 string literal** (`u8` suffix, a `ReadOnlySpan<byte>`, introduced in **C# 11**) with no string materialization. Numbers go straight from the byte span into the CLR primitive via `GetInt32` / `GetDecimal`. We never allocate a `Product`, never box, never build a list for `tags` — we just count. The only allocation is `GetString()` for `name`, which is unavoidable if you actually need the string. `reader.Skip()` fast-forwards over any value subtree (object, array, scalar) you don't care about, which is how you cheaply ignore the 90% of a fat payload you don't read.

Two correctness traps: `Utf8JsonReader` is **forward-only** (no backtracking — if you need a prior value, capture it), and because it's a `ref struct` it **cannot cross an `await`**. To parse data arriving in chunks you use the multi-segment `ReadOnlySequence<byte>` constructor plus `JsonReaderState`, reading what's complete, persisting the `state` and `BytesConsumed`, then resuming on the next buffer — all synchronously, never holding the reader across the I/O `await`.

The writer is the mirror image — push tokens, no intermediate object graph:

```
using System.Buffers;
using System.Text.Json;

public static byte[] WriteProduct(int id, string name,
    decimal price, string[] tags)
{
    var buffer = new ArrayBufferWriter<byte>();
    using var writer = new Utf8JsonWriter(buffer, new
        JsonSerializerOptions { Indented = false });

    writer.WriteStartObject();
    writer.WriteNumber("id"u8, id);
    writer.WriteString("name"u8, name);
    writer.WriteNumber("price"u8, price);
    writer.WriteStartArray("tags"u8);
    foreach (string t in tags)
        writer.WriteStringValue(t);
    writer.WriteEndArray();
    writer.WriteEndObject();

    writer.Flush(); // commit buffered
                   // bytes to the IBufferWriter
    return buffer.WrittenSpan.ToArray();
}
```

`Utf8JsonWriter` is **reusable** — call `Reset(newOutput)` to point it at a fresh `IBufferWriter` / `Stream` and pool the writer instance across requests to avoid reallocating its internal state. Writing UTF-8 property-name literals (`"id"u8`) skips transcoding the key on every call. The writer validates structure by default (matched braces, property-then-value ordering) and throws on misuse; you can relax this with

`JsonSerializerOptions.SkipValidation = true` only if you're certain your token stream is well-formed and need the last few percent of throughput. **.NET 9** added

`JsonSerializerOptions.IndentCharacter` / `IndentSize` and quote-character customization for the rare case where you must match a specific pretty-print style.

Read-only DOM vs mutable DOM: `JsonDocument` and `JsonNode`

When the shape is unknown at compile time, or you only need to peek at a few fields, parse to a DOM rather than a POJO. There are two, and choosing wrong is a common mistake.

`JsonDocument` (with its `JsonElement` struct) is a **read-only**, low-allocation view. It parses the input into a single pooled buffer and exposes `JsonElement` structs that index into it. It is the cheapest way to *read* arbitrary JSON. The catch: `JsonDocument` is `IDisposable` and owns pooled memory; you **must** `using` it, and any `JsonElement` you pull out is only valid for the document's lifetime — capturing a `JsonElement` past the `Dispose` is a use-after-free-style bug (you'll read garbage or throw).

```
using System.Text.Json;

// JsonDocument.Parse takes
//   ReadOnlyMemory<byte>/byte[]/string/Stream/ReadOnlySequence<byte>
// - NOT ReadOnlySpan<byte> (a span can't be stored
//   for the document's lifetime).
byte[] utf8 = """{ "price": 12.50 }"""u8.ToArray();

using JsonDocument doc = JsonDocument.Parse(utf8);
JsonElement root = doc.RootElement;
if (root.TryGetProperty("price"u8, out JsonElement
    priceEl) &&
    priceEl.ValueKind == JsonValueKind.Number)
{
    decimal price = priceEl.GetDecimal();
}
// doc disposed here - do NOT stash 'root' or
//   'priceEl' beyond this scope.
```

`JsonNode` (`System.Text.Json.Nodes`, introduced in .NET 6) is a **mutable**, heap-allocated DOM: `JsonObject`, `JsonArray`, `JsonValue`. You can add, remove, and reassign nodes, then serialize back. It allocates more than `JsonDocument` (every node is an object) and there's no disposal dance, but it's the right tool when you need to *transform* JSON — patch a field, merge documents, inject a property — without binding to a static type.

```

using System.Text.Json;
using System.Text.Json.Nodes;

JsonNode root = JsonNode.Parse("""
    { "id": 7, "name": "Cable", "meta": { "stock": 3 }
    }
    """);

// Read with indexers; the [] navigation is the
// ergonomic win over JsonElement.
int id = (int)root["id"]!;

// Mutate in place.
root["name"] = "USB-C Cable"; //
//      reassign a scalar
root["meta"]!["stock"] = (int)root["meta"]!["stock"]!
    - 1; // decrement nested value
root["tags"] = new JsonArray("usb", "cable"); // add a
//      whole new array property
root.AsObject().Remove("id"); // drop
//      a property

string mutated = root.ToJsonString(new
    JsonSerializerOptions { WriteIndented = true
    });

```

Rule of thumb: **read-only inspection** →

JsonDocument / **JsonElement** (fastest, lowest allocation, mind the lifetime); **transform-and-re-emit** → **JsonNode** (mutable, ergonomic indexers, higher allocation). For *known* shapes, skip the DOM entirely and bind to a type — the DOM is a tool for the dynamic and the unknown.

Polymorphic serialization with [JsonDerivedType]

Serializing a hierarchy behind a base type used to require a custom converter. Since .NET 7, [JsonDerivedType] declares the closed set of subtypes and a **type discriminator** that's written into the JSON, so the deserializer can reconstruct the concrete type. You annotate the base; each derived type gets a discriminator value.

```

using System.Text.Json.Serialization;

[JsonPolymorphic(TypeDiscriminatorPropertyName =
    "$kind")] // default is "$type"
[JsonDerivedType(typeof(CardPayment), "card")]
[JsonDerivedType(typeof(BankTransfer), "bank")]
[JsonDerivedType(typeof(WalletPayment), "wallet")]
public abstract class Payment
{
    public required decimal Amount { get; init; }
}

public sealed class CardPayment : Payment
{
    public required string Last4 { get; init; }
}

public sealed class BankTransfer : Payment
{
    public required string Iban { get; init; }
}

public sealed class WalletPayment : Payment
{
    public required string Provider { get; init; }
}

```

```

Payment payment = new CardPayment { Amount = 49.90m,
    Last4 = "4242" };

// Serializing through the BASE type writes the
// discriminator first:
// {"$kind":"card","Amount":49.90,"Last4":"4242"}
string json = JsonSerializer.Serialize(payment); //
// note: declared type is Payment

// Deserialize back to the base; the runtime picks
// CardPayment from "$kind".
Payment round = JsonSerializer.Deserialize<Payment>
    (json)!;
// round is a CardPayment

```

Three things expert teams get wrong:

- **You must serialize through the base type** for the discriminator to be emitted.

`JsonSerializer.Serialize(payment)` where the *static*

type is `CardPayment` will not write `$kind` — the polymorphism metadata lives on the base. Pass `typeof(Payment)` or a `Payment`-typed variable.

- **The discriminator must be the first property on read.** Because the deserializer is forward-only and must know the concrete type before it materializes any members, `System.Text.Json` requires the `$type` / `$kind` property to appear **first** in the JSON object by default — a discriminator buried later in the object throws `JsonException` (“The metadata property is either not supported by the type or is not in the expected order”). The serializer always writes it first, so round-tripping your own output is safe; the trap is a producer (a JS client, a hand-built payload) that puts it elsewhere. There’s no built-in option to relax the ordering requirement — if you must accept out-of-order discriminators you need a custom converter that buffers the object first.
- **[JsonDerivedType] is a closed set by design.** An incoming discriminator that isn’t declared throws. That’s a feature (it rejects unknown subtypes — a deserialization-gadget defense) but it means you can’t deserialize types you didn’t enumerate. For open hierarchies you still need a custom converter, and you should think hard about whether you *want* arbitrary type instantiation from untrusted input (you usually don’t). The discriminator name choice also matters: prefer stable string tags (`"card"`) over `typeof(...).FullName` , which couples your wire format to your assembly layout.

Polymorphism composes with source-gen: annotate the base type and add `[JsonSerializable(typeof(Payment))]` to your context.

Required members, `init` , and constructor binding

`System.Text.Json` understands modern object-initialization idioms, and getting the binding contract right prevents a class of “silently null” deserialization bugs.

- **required members** (C# 11) are enforced *by the serializer*: if the JSON omits a **required** property, deserialization throws `JsonException`. This turns “missing field” from a latent `null` / `default` into a loud failure at the boundary — exactly where you want it.
- **init-only setters** (C# 9) are honored; the serializer sets them during construction.
- **Parameterized constructors** are bound by **matching constructor parameter names to JSON property names** (case-insensitive when the web defaults are on). For `record` types and immutable classes, this is how the serializer populates `init` / get-only members. When there are multiple constructors, mark the intended one with `[JsonConstructor]`.

```
public sealed class Account
{
    [JsonConstructor]
    public Account(string id, string email) //
        params bind by name to "id"/"email"
    {
        Id = id;
        Email = email;
    }

    public string Id { get; } // get-
        only — set via constructor
    public string Email { get; }
    public required string Region { get; init; } //
        must be present in JSON or throw
    public DateOnly? LastSeen { get; init; } //
        optional; DateOnly is .NET 6
}
```

The interplay is the gotcha: a parameter name that doesn’t match a JSON property (or a property) binds to its *default* rather than throwing, so a renamed constructor parameter can produce a half-constructed object with no error. Combining **required** with constructor binding is the robust pattern — the constructor supplies the immutable identity, **required init** members enforce presence. (Records, C# 9, get a `[JsonConstructor]` -eligible primary constructor for free,

which is why `record` DTOs deserialize cleanly with no extra annotation.)

Streaming large payloads

Buffering a 2 GB JSON array into a `List<T>` to then iterate it is the wrong shape, and `System.Text.Json` gives you two streaming escape hatches.

`JsonSerializer.DeserializeAsyncEnumerable<T>` (.NET 6) reads a **top-level JSON array** and yields elements one at a time as they're parsed off the stream, with bounded memory — you process and discard each element without ever holding the whole array. This is the canonical pattern for ingesting large feeds:

```
using System.Text.Json;

public static async Task ImportAsync(Stream httpBody,
    CancellationToken ct)
{
    var options = JsonSerializerOptions.Web; //
    cached singleton

    // Streams a top-level [ {...}, {...}, ... ]
    // without buffering it all.
    await foreach (Order? order in

        JsonSerializer.DeserializeAsyncEnumerable<Order>(httpBody, options, ct))
    {
        if (order is null) continue;
        await ProcessAsync(order, ct); //
        handle, then the element is GC-eligible
    }
}

static Task ProcessAsync(Order o, CancellationToken
    ct) => Task.CompletedTask;
```

By default `DeserializeAsyncEnumerable` expects the root to be an array (`topLevelValues: false`). .NET 9 added a `topLevelValues: true` overload that instead reads a **root-level stream of whitespace-separated values** (JSON Lines / concatenated JSON), which is how you consume an

`application/jsonl` (newline-delimited JSON) feed where the wire isn't a single array but a sequence of independent documents. (Note this is raw concatenated JSON, not SSE's `data:`-framed `text/event-stream` — you'd strip the SSE framing yourself before feeding the values in.)

.NET 10 went further and added `PipeReader` overloads across the async surface —

`SerializeAsync` / `DeserializeAsync` / `DeserializeAsyncEnumerable` now accept a `System.IO.Pipelines.PipeReader` directly, deserializing with no intermediate `Stream` adapter and effectively no buffering allocation. This plugs straight into `HttpRequest.BodyReader` and pipelines-based servers (Kestrel), and benchmarks land it within a hair of the `Stream` path while removing the adapter overhead:

```
using System.IO.Pipelines;
using System.Text.Json;

// .NET 10: deserialize a streamed array straight off
// the request's PipeReader.
public static async Task IngestAsync(PipeReader body,
    CancellationToken ct)
{
    await foreach (Order? order in
        JsonSerializer.DeserializeAsyncEnumerable<Order>(body, JsonSerializerOptions.Web, ct))
    {
        if (order is not null)
            await ProcessAsync(order, ct);
    }
}
```

The tradeoff with streaming is that you give up random access and the ability to fail the whole batch atomically — you're committed to a forward, element-at-a-time pass. For large, append-style ingestion that's exactly right; for a small payload you'll read into a typed object anyway, the buffering `Deserialize<T>` is simpler and just as fast.

A worked example: a trim-safe, streaming, polymorphic ingestion endpoint

Tying the threads together, here is a complete, AOT-ready ingestion handler that streams a large array of polymorphic `Payment` records off a `PipeReader`, uses a source-generated context (no reflection), applies a custom converter for `Money`, and tallies totals without ever buffering the array. This is the shape of a production hot path.

```

using System.IO.Pipelines;
using System.Text.Json;
using System.Text.Json.Serialization;

// --- Domain -----
[JsonPolymorphic(TypeDiscriminatorPropertyName =
    "$kind")]
[JsonDerivedType(typeof(CardPayment), "card")]
[JsonDerivedType(typeof(BankTransfer), "bank")]
public abstract class Payment
{
    [JsonConverter(typeof(MoneyConverter))]
    public required Money Value { get; init; }
}
public sealed class CardPayment : Payment { public
    required string Last4 { get; init; } }
public sealed class BankTransfer : Payment { public
    required string Iban { get; init; } }

// --- Source-gen context (trim/AOT-safe, cached
// metadata) -----
[JsonSourceGenerationOptions(PropertyNamingPolicy =
    JsonKnownNamingPolicy.CamelCase)]
[JsonSerializable(typeof(Payment))] // base type -
// discovers derived types
public partial class PaymentContext :
    JsonSerializerContext { }

// --- Handler -----
public static class PaymentIngest
{
    // Build options ONCE, bind the generated
    // resolver, freeze.
    private static readonly JsonSerializerOptions
        Options = BuildOptions();

    private static JsonSerializerOptions
        BuildOptions()
    {
        var o = new
            JsonSerializerOptions(JsonSerializerDefaults.Web)
        {
            TypeInfoResolver = PaymentContext.Default,
            // no reflection
            Converters = { new MoneyConverter() },
        };
        o.MakeReadOnly();
        // fail fast on late mutation
        return o;
    }
}

```

```

    }

    public static async Task<decimal>
        SumAsync(PipeReader body, CancellationToken
            ct)
    {
        decimal total = 0m;

        // .NET 10 PipeReader overload + .NET 6
        // streaming enumerable.
        await foreach (Payment? p in

            JsonSerializer.DeserializeAsyncEnumerable<Paym
                ent>(body, Options, ct))
        {
            if (p is null) continue;

            // Polymorphic pattern match on the
            // reconstructed concrete type.
            decimal amount = p switch
            {
                CardPayment card =>
                    card.Value.Amount, // could weight by type
                BankTransfer bank =>
                    bank.Value.Amount,
                _ => p.Value.Amount,
            };
            total += amount;
        }

        return total;
    }
}

```

Walking the design decisions: the options instance is **static, resolver-bound, and frozen** — one metadata build, reused forever, no reflection (the `PaymentContext` resolver makes it AOT-safe end to end). Streaming via `DeserializeAsyncEnumerable` over the `PipeReader` keeps memory **O(1) in the array length** — we never hold more than one `Payment` at a time, and the `PipeReader` path (.NET 10) avoids a `Stream` adapter. Polymorphism is reconstructed from the `$kind` discriminator with a **closed**, declared subtype set, so a hostile payload can't instantiate an arbitrary type. The `MoneyConverter` enforces the decimal-string wire format at the leaf. `required` members guarantee that a `Payment` missing its `Value`, `Last4`, or `Iban`

throws at the boundary rather than producing a half-null object downstream.

Pitfalls, in one place

A consolidated checklist of the traps that bite experienced teams:

- **Reflection mode is trim/AOT-unsafe.** It works under the JIT, then `NotSupportedException`s or silently empties objects once you trim or publish AOT. If you target NativeAOT or trim a container, use a `JsonSerializerContext`. Heed IL2026/IL3050.
- **Default matching is case-sensitive and emits PascalCase.** Web clients want camelCase and case-insensitive binding — opt in via `JsonSerializerDefaults.Web` (or `JsonSerializerOptions.Web`, .NET 9). The “works in the controller, not the console app” mystery is always this.
- **Never construct `JsonSerializerOptions` per call.** The first use builds and freezes a per-instance metadata cache; a fresh instance each time throws it away and re-pays full contract/reflection cost. Make it `static readonly` (and `MakeReadOnly()` it, .NET 8, to fail fast).
- **`Utf8JsonReader` is a `ref struct`** — it cannot cross an `await`, be captured by a lambda, stored in a field, or live inside an async/iterator method. Parse synchronously; for chunked input use `ReadOnlySequence<byte>` + `JsonReaderState`.
- **`JsonDocument` owns pooled memory.** `using` it, and don't let `JsonElement`s escape its scope. For mutation use `JsonNode` (higher allocation, but mutable and disposal-free).
- **Polymorphism only writes the discriminator when serialized through the base type**, and the derived-type set is closed by design — declared types only, which is both a constraint and a security feature.
- **Use the generic `JsonStringEnumConverter<TEnum>` (.NET 8)**, not the reflection-based non-generic one, anywhere AOT matters.

- **Converters must leave the reader on the last token of their value** — over- or under-reading corrupts subsequent parsing.

`System.Text.Json` rewards understanding its layers. Reach for `JsonSerializer` and a cached, web-configured `JsonSerializerOptions` for the 90% case; add a `JsonSerializerContext` the moment trimming or AOT enters the picture; write a `JsonConverter<T>` for domain types with bespoke wire formats; drop to `Utf8JsonReader` / `Utf8JsonWriter` only when a profiler proves the serializer's overhead is your bottleneck; and pick `JsonDocument` vs `JsonNode` by whether you're reading or transforming. Do that, and you get a JSON stack that's fast, allocation-lean, and ready for the AOT future the runtime is steering toward.

26 · Dates, Times, and Ambient State

Time is the most deceptively simple domain in the entire BCL. The naive code (`DateTime.Now` , a `Stopwatch` , a hand-rolled timer) compiles, passes the demo, and then quietly fails in production: tests flake, scheduled jobs drift across DST boundaries, and “expires in 30 minutes” means something different on a UTC server than on a developer’s laptop. This chapter is about choosing the *right* temporal type for each job, and — just as importantly — treating the clock as an injected dependency rather than ambient global state, using the `TimeProvider` abstraction introduced in .NET 8.

The four temporal types, and when each is correct

The BCL gives you four value types to represent points and components of civil time. Picking the wrong one is the root cause of most date bugs.

`DateTime` (since .NET Framework 1.0) is a date *and* time of day, plus a three-state `Kind` tag (`Utc` , `Local` , `Unspecified`). It carries **no offset**, which is the source of nearly all of its pitfalls. `DateTimeOffset` (since .NET 3.5) is a date, a time, *and* an explicit UTC offset — it unambiguously identifies an instant on the global timeline. `DateOnly` and `TimeOnly` (both introduced in .NET 6) finally split the two independent concepts that `DateTime` had been forced to conflate for two decades: a calendar date with no time component, and a wall-clock time of day with no date.

The decision tree is short:

- **A precise instant** (when did this event happen, when does this token expire) → `DateTimeOffset` , stored and computed in UTC.
- **A calendar date** (date of birth, invoice due date, a public holiday) → `DateOnly` .

- **A recurring time of day** (store opens at 09:00, a cron-like “run at 14:30”) → `TimeOnly`.
- `DateTime` → only at interop boundaries you do not control (legacy APIs, some ORMs, `DateTime`-typed database columns), and even then convert in and out as early as possible.

The reason `DateTimeOffset` is “usually right” for instants is that it cannot be silently misinterpreted. A `DateTime` value of `2026-03-29 02:30:00` with `Kind == Unspecified` is *meaningless* without an external convention — and in central Europe that wall-clock time does not even exist on the DST spring-forward night. A `DateTimeOffset` of `2026-03-29 02:30:00+01:00` is a real, ordered, comparable instant.

It is worth being precise about what `DateTimeOffset` does and does *not* capture, because the name oversells it. It records the offset from UTC at that moment (`+01:00`), which is enough to pin the instant on the global timeline and to compare, sort, and subtract values correctly. It does **not** record the *time zone* — the offset alone cannot tell you whether the value will be `+01:00` or `+02:00` six months later, because it doesn’t know the DST rules. When future-dated, zone-aware behavior matters (a recurring meeting “at 09:00 Prague time, every Monday, forever”), the durable representation is the pair (*local wall time*, *IANA zone id*), resolved to a concrete `DateTimeOffset` at the moment you need an instant — never a single offset frozen at creation time. For a *past* event, the `DateTimeOffset` is sufficient and ideal; for a *future* recurring obligation, store the zone.

DateOnly and TimeOnly: arithmetic without the time-of-day baggage

Before .NET 6, representing “due on the 14th” meant a `DateTime` at midnight, and you spent your career defending that midnight from accidental `.AddHours`, time-zone conversion, and serialization that smuggled in a `T00:00:00Z`. `DateOnly` removes the time component from the type system entirely.

```

using System;

// DateOnly arithmetic (.NET 6). No hidden midnight,
// no Kind, no offset.
DateOnly invoiceDate = new(2026, 5, 30);
DateOnly dueDate = invoiceDate.AddDays(30);
// 2026-06-29

// First of next month, then step back one day to land
// on the last day of THIS
// month – robust across 28/29/30/31-day months
// without a length lookup.
DateOnly firstOfNextMonth = new
    DateOnly(invoiceDate.Year, invoiceDate.Month,
        1).AddMonths(1);
DateOnly endOfMonth = firstOfNextMonth.AddDays(-1); //
    2026-05-31

int daysOutstanding = dueDate.DayNumber -
    invoiceDate.DayNumber; // 30
DayOfWeek dow = dueDate.DayOfWeek;
// Monday

// TimeOnly is a time of day in {00:00:00, 24:00:00}.
// Arithmetic wraps the clock.
TimeOnly open = new(9, 0);
TimeOnly close = new(17, 30);
TimeSpan tradingHours = close - open;
// 08:30:00

// AddHours wraps past midnight and reports how many
// days it crossed.
TimeOnly late = open.AddHours(20, out int
    wrappedDays); // 05:00, wrappedDays == 1
bool isOpenAt(TimeOnly t) => t.IsBetween(open, close);
// half-open [open, close)

```

`DateOnly.DayNumber` (days since 0001-01-01) gives you allocation-free integer date subtraction, which is both faster and clearer than `(d1 - d2).Days` on `DateTime`. `TimeOnly.IsBetween` uses a half-open interval and — crucially — handles the wrap-around case (`IsBetween(new(22,0), new(2,0))` correctly treats a window that crosses midnight), which a naive `t >= start && t < end` comparison gets wrong.

Both types are also `readonly struct`s the size of their underlying primitive (`DateOnly` is a four-byte day number; `TimeOnly` is an eight-byte tick count), so they pass by value cheaply, never

allocate, and compare and hash without boxing. That makes them excellent dictionary keys and set members where `DateTime` would drag along an irrelevant time-of-day in the equality contract. A subtle correctness win comes from the type system itself: a method that takes a `DateOnly` *cannot* be handed a value carrying a stray `13:47:02.331`, so an entire category of “why is my date comparison failing — oh, one side has a time component” bugs simply cannot be expressed. Choosing the narrowest type that models your domain is a correctness tool, not just a stylistic one.

Combine them only at the edge, and only with an explicit kind:

```
DateOnly d = new(2026, 5, 30);
TimeOnly t = new(14, 15);
DateTime combined = d.ToDateTime(t, DateTimeKind.Utc);
// be explicit about Kind
```

One caveat worth stating plainly: `DateOnly` and `TimeOnly` are *not* directly mappable by every data-access stack or serializer version you’ll meet. `System.Text.Json` gained built-in converters in .NET 7; older EF Core providers needed value converters. Before adopting them in a persistence model, confirm your provider round-trips them natively rather than via `DateTime`.

DateTime.Kind: the pit of failure

`DateTime.Kind` is a tag, not a guarantee, and the arithmetic does not respect it. This is the single most common source of “works on my machine” time bugs.

```

using System;

DateTime unspecified = new(2026, 5, 30, 12, 0, 0);
    // Kind == Unspecified
DateTime utc        = DateTime.UtcNow;
    // Kind == Utc
DateTime local      = DateTime.Now;
    // Kind == Local

// Pitfall 1: ToUniversalTime on an Unspecified value
// ASSUMES it is Local.
DateTime guessUtc = unspecified.ToUniversalTime();
    // silently uses machine TZ

// Pitfall 2: arithmetic ignores Kind. Adding 24h does
// NOT account for a DST
// transition – DateTime has no offset to adjust. This
// "loses" or "gains" an hour
// relative to civil wall time across a DST boundary.
DateTime tomorrow = local.AddHours(24);

// Pitfall 3a: equality ignores Kind and compares raw
// ticks. These two have the
// SAME wall-clock ticks but different Kind, yet ==
// reports them EQUAL – it never
// reconciles the Kinds, so a Utc value and an
// Unspecified value with identical
// clock fields compare equal even though they may
// denote different instants.
DateTime aUtc    = new(2026, 5, 30, 12, 0, 0,
    DateTimeKind.Utc);
DateTime bUnspec = new(2026, 5, 30, 12, 0, 0,
    DateTimeKind.Unspecified);
bool equalDespiteKind = aUtc == bUnspec;           //
    true – Kind is ignored

// Pitfall 3b: trying to "fix" it by normalizing
// through ToUniversalTime is
// machine-dependent, because the Unspecified side is
// assumed to be Local.
bool same = aUtc == bUnspec.ToUniversalTime(); //
    result depends on machine TZ – fragile

```

The deeper problem is that `Unspecified` is the *default Kind* for any `DateTime` you parse, deserialize from JSON without `DateTimeOffset`, or read from a `datetime` / `datetime2` SQL column. The value arrives stripped of context, and the first method that needs context (`ToUniversalTime`, `ToLocalTime`) fabricates it from `TimeZoneInfo.Local`. On a UTC-configured

CI runner that fabrication happens to be a no-op; on a developer machine in `Europe/Prague` it shifts by one or two hours. Tests that pass in CI and fail locally (or vice versa) are the signature of this bug.

`DateTimeOffset` sidesteps all three pitfalls because it stores the offset alongside the ticks. Comparison and equality (`==`, `CompareTo`, `EqualsExact`) operate on the underlying UTC instant, so two `DateTimeOffset` values for the same moment in different zones *are* equal under `==`. Make `DateTimeOffset` your default at every boundary you control, and reserve `DateTime` for the columns and APIs that demand it.

The clock as a dependency: `TimeProvider` (.NET 8)

`DateTime.UtcNow` and `DateTimeOffset.UtcNow` are *ambient global state*. Every call reaches out to a process-wide, machine-controlled source of truth that you cannot observe, control, or freeze from a test. Code that calls them directly is non-deterministic by construction: you can't write "given it's 23:59, when I add 2 minutes, then it rolls to tomorrow" without literally waiting for 23:59 or resorting to brittle mocking shims.

`TimeProvider`, introduced in .NET 8 (`System.TimeProvider`, in the core libraries — no NuGet package needed), is the abstraction that fixes this. It is an abstract base class with a small, deliberately complete surface:

- `GetUtcNow()` → `DateTimeOffset` — the current instant. Note it returns an *offset*, steering you toward the right type.
- `LocalTimeZone` → `TimeZoneInfo` — the provider's notion of "local," overridable in tests instead of reading `TimeZoneInfo.Local`.
- `GetTimestamp()` → `long` and `GetElapsedTime(long)` / `GetElapsedTime(long, long)` → `TimeSpan` — a high-resolution, monotonic stopwatch decoupled from wall time.
- `TimestampFrequency` → `long` — ticks per second for the high-res timer.

- `CreateTimer(TimerCallback, object?, TimeSpan, TimeSpan) → ITimer` — a fully abstracted, fakeable timer.

`TimeProvider.System` is the singleton that delegates to the real OS clock. You inject *that* in production and a fake in tests. The rule is mechanical: **never call `DateTime.UtcNow`, `DateTimeOffset.UtcNow`, `DateTime.Now`, or `new Stopwatch()` in domain code; take a `TimeProvider` and call it instead.**

Registering it for DI is a one-liner:

```
using Microsoft.Extensions.DependencyInjection;

// In Program.cs / composition root.
// TimeProvider.System is a process-wide
// singleton.
services.AddSingleton(TimeProvider.System);
```

Worked example: a testable token-expiry service

Here is the canonical pattern end to end. The service computes and validates an expiry; it depends only on `TimeProvider`, never on the ambient clock. Note the C# 12 (.NET 8) **primary constructor** capturing the dependency, and the C# 14 (.NET 10) **field keyword** used to add validation to an auto-property without declaring a backing field by hand.

```

using System;

namespace Auth;

public sealed record AccessToken(string Value,
    DateTimeOffset IssuedAt, DateTimeOffset
    ExpiresAt)
{
    public bool IsExpiredAt(DateTimeOffset now) => now
        >= ExpiresAt;
}

// Primary constructor (C# 12) injects the clock; no
// field declaration boilerplate.
public sealed class TokenService(TimeProvider
    timeProvider)
{
    // field keyword (C# 14): validate without a hand-
    // written backing field.
    public TimeSpan Lifetime
    {
        get;
        init => field = value > TimeSpan.Zero
            ? value
            : throw new
                ArgumentOutOfRangeException(nameof(value),
                    "Lifetime must be positive.");
    } = TimeSpan.FromMinutes(30);

    public AccessToken Issue(string value)
    {
        DateTimeOffset issuedAt =
            timeProvider.GetUtcNow(); // NOT
            DateTimeOffset.UtcNow
        return new AccessToken(value, issuedAt,
            issuedAt + Lifetime);
    }

    public bool IsValid(AccessToken token)
        =>
            !token.IsExpiredAt(timeProvider.GetUtcNow());
}

```

Everything temporal funnels through

`timeProvider.GetUtcNow()`, so the entire expiry boundary is now controllable. The test uses `FakeTimeProvider` from the `Microsoft.Extensions.TimeProvider.Testing` NuGet package (the type lives in the `Microsoft.Extensions.Time.Testing` namespace — note the package name and namespace differ).

`Advance(TimeSpan)` moves the synthetic clock forward by an exact amount; `SetUtcNow(DateTimeOffset)` jumps it to an absolute instant.

```
using System;
using Microsoft.Extensions.Time.Testing; // package:
    Microsoft.Extensions.TimeProvider.Testing
using Xunit;

public class TokenServiceTests
{
    [Fact]
    public void
        Token_is_valid_right_up_to_the_boundary_then_expires()
    {
        // Pin the synthetic clock to a fixed, fully-
        // specified instant.
        var start = new DateTimeOffset(2026, 5, 30,
            12, 0, 0, TimeSpan.Zero);
        var time = new FakeTimeProvider(start);
        var sut = new TokenService(time) { Lifetime =
            TimeSpan.FromMinutes(30) };

        AccessToken token = sut.Issue("abc");
        Assert.Equal(start.AddMinutes(30),
            token.ExpiresAt);

        // 1 second before expiry: still valid.
        time.Advance(TimeSpan.FromMinutes(30) -
            TimeSpan.FromSeconds(1));
        Assert.True(sut.IsValid(token));

        // Cross the boundary deterministically. No
        // Thread.Sleep, no real time elapsed.
        time.Advance(TimeSpan.FromSeconds(1));
        Assert.False(sut.IsValid(token));
    }
}
```

This test runs in microseconds, never flakes, and asserts the *exact* boundary condition — the one that real-time tests can only approximate by sleeping and hoping. That is the entire value proposition of `TimeProvider` : it turns time from an uncontrollable input into a parameter.

Contrast this with the pre-.NET-8 alternatives. The “static via a delegate” trick (`Func<DateTimeOffset> _now` injected into the

constructor) gives you testability for *the clock reading* but nothing for timers, elapsed time, or local zone — you end up inventing four parallel seams. Hand-rolled `IClock` interfaces proliferated across the ecosystem (NodaTime's `IClock`, countless bespoke ones), each incompatible with the next, so a library exposing `IClock` couldn't be wired to an app using `ISystemClock`. `TimeProvider` ends that fragmentation: it is the *single* abstraction in the BCL that every new library targets, so a `FakeTimeProvider` you construct in a test drives your code, your dependencies, and the framework's own timer-based machinery uniformly. When you adopt it, delete the bespoke clock interfaces rather than adapting them.

One migration caveat: if you have existing code calling `DateOffset.UtcNow` scattered across a large codebase, resist a mechanical find-and-replace to `timeProvider.GetUtcNow()` in one pass. The dependency has to be *threaded* — constructors gain a parameter, factories gain an argument, and static helpers must become instance methods or take the provider explicitly. Do it leaf-first (start at the classes that read the clock, push the dependency up to their callers) and let the compiler errors guide you; a half-threaded migration where some paths use the injected clock and others still hit the ambient one is worse than not starting, because tests will pass while production reads two different clocks.

A note on `FakeTimeProvider`'s `AutoAdvanceAmount` property: by default the fake clock is frozen and only moves when you call `Advance / SetUtcNow`. If you set `AutoAdvanceAmount` to a non-zero `TimeSpan`, every read of `GetUtcNow()` / `GetTimestamp()` advances the clock by that amount, which is handy for code that expects strictly increasing timestamps but should be used sparingly — it reintroduces a form of nondeterminism if your assertions depend on the number of reads.

High-resolution timing: `GetTimestamp` / `GetElapsedTime` vs the wall clock

Measuring a *duration* is a different problem from reading the *time of day*, and conflating them is a classic mistake. The wall clock (`GetUtcNow`) can jump backward when NTP corrects it or when an operator changes the system time; subtracting two wall-clock readings to time an operation can therefore yield a negative or wildly wrong duration. For elapsed-time measurement you want a *monotonic* high-resolution counter that only ever moves forward.

Before .NET 8 the answer was

`System.Diagnostics.Stopwatch` . It still works and is fine for ad-hoc microbenchmarks, but it is *ambient* in the same way `DateTime.UtcNow` is — you can't fake it.

`TimeProvider.GetTimestamp()` plus `GetElapsedTime()` give you the same monotonic counter behind a fakeable abstraction (both added in .NET 8). The timestamp is an opaque `long` in `TimestampFrequency` units; you never interpret it directly, you only diff it.

```
using System;

public sealed class LatencyProbe(TimeProvider
    timeProvider)
{
    public (T Result, TimeSpan Elapsed) Measure<T>
        (Func<T> work)
    {
        long start = timeProvider.GetTimestamp(); //
        // monotonic, high-resolution
        T result = work();
        // Overload that takes only the start
        // timestamp implicitly uses "now".
        TimeSpan elapsed =
            timeProvider.GetElapsedTime(start);
        return (result, elapsed);
    }
}
```

`GetElapsedTime(start)` is shorthand for `GetElapsedTime(start, GetTimestamp())` . In a hot loop

where you want to time many segments against a single base, capture one `GetTimestamp()` at the start and call the two-argument overload, avoiding repeated work. Three rules:

- Use `GetTimestamp` / `GetElapsedTime` (or `Stopwatch`) for **durations**; use `GetUtcNow` for **instants**. Never subtract two `GetUtcNow()` values to time code.
- The timestamp is process-monotonic but has **no meaning across processes or machines** — don't log it, don't persist it, don't compare it to another process's timestamps.
- `Stopwatch.GetTimestamp()` / `Stopwatch.GetElapsedTime(long)` (the static, allocation-free form added in .NET 7) is the right tool when you specifically *don't* want testability overhead — e.g. inside a library's internal benchmark harness. But for application code that should be unit-testable, prefer the `TimeProvider` instances so a `FakeTimeProvider` can drive them.

Fakeable timers: `ITimer` from `TimeProvider`

The same ambient-state problem afflicts

`System.Threading.Timer`, `PeriodicTimer`, and `Task.Delay`: a test that exercises “fire every 5 minutes” logic would have to actually wait. `TimeProvider.CreateTimer` returns an `ITimer` (both .NET 8) whose firing is driven by the *provider's* clock — so with a `FakeTimeProvider`, advancing time fires the timer synchronously and deterministically.

```

using System;
using System.Threading;

public sealed class Heartbeat : IDisposable
{
    private readonly ITimer _timer;
    public int Beats { get; private set; }

    public Heartbeat(TimeProvider timeProvider,
        TimeSpan interval)
    {
        // dueTime and period: fire first after
        // `interval`, then every `interval`.
        _timer = timeProvider.CreateTimer(
            callback: _ => Beats++,
            state: null,
            dueTime: interval,
            period: interval);
    }

    public void Dispose() => _timer.Dispose();
}

```

```

using System;
using Microsoft.Extensions.Time.Testing;
using Xunit;

public class HeartbeatTests
{
    [Fact]
    public void Fires_once_per_interval_when_time_advances()
    {
        var time = new FakeTimeProvider();
        using var hb = new Heartbeat(time,
            TimeSpan.FromSeconds(5));

        Assert.Equal(0, hb.Beats);

        time.Advance(TimeSpan.FromSeconds(5));    //
            first fire
        Assert.Equal(1, hb.Beats);

        time.Advance(TimeSpan.FromSeconds(12));
        // crosses two more periods (10s, 15s)
        Assert.Equal(3, hb.Beats);
    }
}

```

`FakeTimeProvider.Advance` fires every timer whose due time falls within the advanced interval, in order, on the calling thread — so the assertions above are exact, not racy. Trace the schedule explicitly: with a 5-second period the timer fires at $t=5s$ (first beat), and the second `Advance` jumps from $t=5s$ to $t=17s$, crossing the $t=10s$ and $t=15s$ firing points — two more beats, for three total. This kind of precise schedule reasoning is exactly what real-time tests cannot do, because thread scheduling jitter blurs the boundaries; with the fake provider the firings are deterministic and synchronous, so you can assert against the exact count.

For genuinely periodic async loops, `PeriodicTimer` is still the ergonomic choice in production, and since .NET 8 it accepts a `TimeProvider` in its constructor (`new PeriodicTimer(period, timeProvider)`), so `await timer.WaitForNextTickAsync()` can be driven by a `FakeTimeProvider` too. Reach for the lower-level `ITimer` from `CreateTimer` when you need a callback-style timer (no `await` loop) or fine control over distinct `dueTime` and `period` values; reach for `PeriodicTimer` when you want a clean `while (await ...)` consumer loop. Both are testable, which is the point — the moment a timer's cadence affects observable behavior, it must not be ambient.

There is a subtle ordering guarantee worth internalizing: a single `Advance` that spans multiple firing points invokes the callbacks *during* the `Advance` call, before it returns, in timestamp order. This means a callback that itself reads `GetUtcNow()` sees the clock value *at its own firing instant*, not the post-`Advance` value — the fake provider replays the timeline rather than teleporting to the end. Code under test that timestamps each tick therefore records the correct per-tick instants, which is what you want when verifying, say, that a rate limiter spaced its events correctly.

A production-grade background service should take the `TimeProvider` from DI and use `CreateTimer`, so the whole scheduler becomes unit-testable:

```

using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;

public sealed class CleanupService(TimeProvider
    timeProvider) : BackgroundService
{
    protected override Task
        ExecuteAsync(CancellationToken stoppingToken)
    {
        var timer = timeProvider.CreateTimer(
            _ => { /* enqueue cleanup work */,
                state: null,
                dueTime: TimeSpan.FromMinutes(1),
                period: TimeSpan.FromMinutes(15));

            stoppingToken.Register(() => timer.Dispose());
            return Task.CompletedTask;
        }
    }
}

```

Time zones: conversions, IANA vs Windows ids

A `DateTimeOffset` knows its offset but *not* its zone — `+01:00` could be Prague in winter or Lagos year-round. Converting an instant to a particular region's wall-clock time, with DST handled correctly, is the job of `TimeZoneInfo`.

The cross-platform identifier headache is that Windows historically used its own zone ids (`"Central European Standard Time"`) while the rest of the world uses **IANA** ids (`"Europe/Prague"`). Since **.NET 6**, `TimeZoneInfo.FindSystemTimeZoneById` accepts *either* form on *any* platform (Windows resolves IANA ids via ICU), and from **.NET 8** it returns a cached instance. When you must map explicitly — e.g. to store a portable id — use `TimeZoneInfo.TryConvertIanaIdToWindowsId` and `TryConvertWindowsIdToIanaId` (both .NET 6, ICU-dependent). The strong recommendation is to **standardize on IANA ids in your data**, because they are stable, unambiguous, and portable; convert to Windows ids only at a Windows-specific boundary.

```

using System;

// Resolve a zone by IANA id – works on Windows,
// Linux, and macOS since .NET 6.
TimeZoneInfo prague =
    TimeZoneInfo.FindSystemTimeZoneById("Europe/Pr
ague");

// An instant in UTC...
DateTimeOffset instant = new(2026, 7, 1, 12, 0, 0,
    TimeSpan.Zero);

// ...converted to that zone's wall-clock time, DST
// applied automatically.
DateTimeOffset pragueLocal =
    TimeZoneInfo.ConvertTime(instant, prague); //
    14:00 +02:00 (CEST)

// Round-trip: same instant, different presentation.
DateTimeOffset back = pragueLocal.ToUniversalTime();
// 12:00 +00:00
bool sameInstant = back == instant;
// true

// Portable id mapping, only at a Windows-specific
// edge.
if
    (TimeZoneInfo.TryConvertIanaIdToWindowsId("Eur
ope/Prague", out string? winId))
{
    // winId == "Central European Standard Time"
}

// DST awareness is built in: this date is the EU
// spring-forward day.
DateTime springForward = new(2026, 3, 29, 2, 30, 0);
// 02:30 does NOT exist in CET that night
bool invalid = prague.IsInvalidTime(springForward);
// true – handle before converting
bool ambiguous = prague.IsAmbiguousTime(new
    DateTime(2026, 10, 25, 2, 30, 0)); // fall-
// back day

```

`IsInvalidTime` and `IsAmbiguousTime` are the guardrails for the two DST pathologies: the spring-forward gap (a wall-clock time that never occurs) and the fall-back overlap (a wall-clock time that occurs twice). Convert *instants* (UTC `DateTimeOffset`) into local presentation and you avoid the gap entirely; the danger lives only when you start from a user-en-

tered local `DateTime` with no offset and must decide which UTC instant it meant.

When you do have to resolve an ambiguous local time,

`TimeZoneInfo.GetUtcOffset(DateTime)` returns the offset the framework *chooses* (the standard-time offset, by convention, for an ambiguous value), while

`GetAmbiguousTimeOffsets(DateTimeOffset)` hands you *both* candidate offsets so you can apply a business rule — for an overlap on the fall-back night, a logistics system might deliberately pick the earlier (DST) instant. There is also a performance angle:

`FindSystemTimeZoneById` does real work — a registry or ICU lookup the first time — and although .NET 8 caches the result, resolving the same id in a tight loop still pays a dictionary lookup and a lock. In hot paths, resolve the `TimeZoneInfo` once and hold the reference, or stash frequently-used zones in a

`FrozenDictionary<string, TimeZoneInfo>`

(`System.Collections.Frozen`, .NET 8) keyed by IANA id for allocation-free, read-optimized lookup. Never call

`FindSystemTimeZoneById` per request when the set of zones is small and known at startup.

Parsing, formatting, culture, and round-tripping

Serialization and logging are where carefully-typed temporal values get flattened back into strings — and culture turns that into a minefield. `ToString()` and `Parse` without an explicit

`IFormatProvider` use the *current thread culture*, so

`DateTime.Parse("01/02/2026")` is January 2nd in the US and February 1st in much of Europe. For any machine-to-machine string, this is a latent bug.

Two rules cover almost everything:

1. For **persistence, APIs, and logs**, use the **round-trip format specifier "O"** (ISO 8601) with `CultureInfo.InvariantCulture`. It preserves sub-second precision and, for `DateTimeOffset` / `DateTime`, the offset/ `Kind`, so a value survives a string round-trip byte-for-byte.

2. For **user-facing display**, use a specific `CultureInfo` chosen by the user's locale — never the ambient `CurrentCulture` by accident.

```
using System;
using System.Globalization;

DateTimeOffset now = new(2026, 5, 30, 14, 15, 30, 123,
    TimeSpan.FromHours(2));

// Round-trip ("0") – ISO 8601 with offset, invariant.
// The format to log and persist.
string iso = now.ToString("0",
    CultureInfo.InvariantCulture);
// "2026-05-30T14:15:30.1230000+02:00"

DateTimeOffset parsed = DateTimeOffset.ParseExact(
    iso, "0", CultureInfo.InvariantCulture,
    DateTimeStyles.RoundtripKind);
bool exact = parsed.EqualsExact(now); // true –
    instant AND offset preserved

// DateOnly / TimeOnly also honor "0" for unambiguous
// round-tripping.
DateOnly d = new(2026, 5, 30);
string dIso = d.ToString("0",
    CultureInfo.InvariantCulture); // "2026-05-30"

// Culture-sensitive PARSE is the trap. Always pass
// the provider explicitly.
var de = CultureInfo.GetCultureInfo("de-DE");
DateTime german = DateTime.Parse("30.05.2026", de);
// 30 May, correct for de-DE

// Use the invariant culture for anything that crosses
// a wire or a file.
bool ok = DateTime.TryParse(
    "2026-05-30T12:00:00Z",
    CultureInfo.InvariantCulture,
    DateTimeStyles.RoundtripKind, out DateTime utc);
// Kind == Utc, ok == true
```

`DateTimeStyles.RoundtripKind` is what preserves the `Z` /offset semantics on parse — omit it and a trailing `Z` may be honored while a bare value lands as `Unspecified`. For `DateTimeOffset`, prefer `EqualsExact` over `==` when you want to assert that *both* the instant and the textual offset round-tripped, since `==` compares only the underlying instant.

A performance footnote: `ToString("0", ...)` and `ParseExact` allocate a string and walk a format, which is negligible for logging but can show up in tight serialization loops. `System.Text.Json` (with its own internal ISO 8601 writers since .NET Core 3.0, refined through .NET 9/10) formats `DateTimeOffset` far more efficiently than round-tripping through `ToString`; let the serializer handle temporal types directly rather than pre-converting them to strings. If you do need the absolute fastest path, the `TryFormat(Span<char>, ...)` overloads write into a stack-allocated buffer with zero heap allocation — relevant in high-throughput logging or protocol code.

A consolidated extension surface (C# 14 extension members)

C# 14 (.NET 10) **extension members** (declared inside an `extension` block in a static class) let you bundle helpers — including extension *properties* — onto existing types without the old one-method-at-a-time syntax. This is a clean way to give your codebase a small, discoverable vocabulary over `TimeProvider` and the temporal types. Below, an extension property and method hang off `TimeProvider` itself.

```

using System;

public static class TimeExtensions
{
    extension(TimeProvider time)
    {
        // Extension PROPERTY (C# 14): today's
        // calendar date in the provider's UTC clock.
        public DateOnly UtcToday =>
            DateOnly.FromDateTime(time.GetUtcNow().UtcDate
            Time);

        // Extension method: instant projected into a
        // named IANA zone.
        public DateTimeOffset NowIn(string ianaZoneId)
            => TimeZoneInfo.ConvertTime(
                time.GetUtcNow(),

                TimeZoneInfo.FindSystemTimeZoneById(ianaZoneId
                ));
    }
}

// Usage reads naturally and stays fully testable
// (drive it with FakeTimeProvider):
// DateOnly today = timeProvider.UtcToday;
// DateTimeOffset prague =
//     timeProvider.NowIn("Europe/Prague");

```

Because these extensions resolve through the injected `TimeProvider`, they inherit its testability for free — a `FakeTimeProvider` makes `UtcToday` and `NowIn` deterministic exactly like the underlying `GetUtcNow()`.

Summary and rules of thumb

Time is ambient state in disguise, and the cure is the same as for any other dependency: inject it. Concretely:

- **Default to `DateTimeOffset` for instants,**
`DateOnly` / `TimeOnly` (.NET 6) for dates and times-of-day,
 and quarantine `DateTime` to interop edges. Treat
`DateTime.Kind == Unspecified` as a defect waiting to
 happen.
- **Never call `DateTime.UtcNow` / `DateTime.Now` / `new Stopwatch()` in domain code.** Take a `TimeProvider`

(.NET 8), register `TimeProvider.System` in DI, and call `GetUtcNow`, `GetTimestamp` / `GetElapsedTime`, and `CreateTimer`. Test with `FakeTimeProvider.Advance` for deterministic boundary crossing.

- **Distinguish durations from instants.** Monotonic `GetTimestamp` / `GetElapsedTime` for “how long did this take”; `GetUtcNow` for “when did this happen.” Subtracting wall-clock reads to time code is a bug.
- **Store IANA time-zone ids**, resolve with `FindSystemTimeZoneById`, convert with `TimeZoneInfo.ConvertTime`, and guard DST with `IsValidTime` / `IsAmbiguousTime`.
- **Round-trip with "0" and `CultureInfo.InvariantCulture`** on every wire and file; reserve culture-sensitive formatting for the display layer and pass the `IFormatProvider` explicitly, always.

When *not* to reach for `TimeProvider`: trivial scripts, throwaway benchmarks, or library internals where the abstraction’s indirection buys nothing and you’ve consciously decided the code will never be unit-tested against the clock. Everywhere else — anywhere a date, a timeout, an expiry, or a schedule influences behavior you care about verifying — the injected clock pays for itself the first time a test crosses a boundary at exactly the right microsecond instead of sleeping and praying.

27 · Error Handling, Validation, and Resilience

Error handling is where the gap between code that works on your machine and code that survives production becomes a chasm. This chapter treats exceptions as a design surface — not an afterthought — and walks the full ladder from a single guard clause through stack-trace preservation, cancellation semantics, result modeling, and process-wide resilience pipelines. The recurring theme: an exception is a contract with the caller, and most of the bugs in this space come from breaking that contract silently.

When to throw, and the never-catch types

A throw is the right tool when a method cannot fulfill its post-condition and the caller almost certainly cannot continue down its current path. It is the *wrong* tool for outcomes that are routine and expected — a cache miss, a validation rejection on user input, a “no rows matched.” Those belong in return values (see *Result modeling* below). The litmus test: if a reasonable caller would wrap the call in `try/catch` and treat the catch as normal flow, you should have returned a value instead.

The .NET exception hierarchy gives you three tiers to reason about. `SystemException` and its subtypes (`InvalidOperationException`, `ArgumentException`, `NotSupportedException`) signal programmer error or contract violation — usually bugs to be fixed, not handled. Custom domain exceptions deriving from `Exception` signal recoverable application-level failures. And then there are the types you must **never catch to swallow**: `StackOverflowException` (uncatchable since .NET 2.0 — it tears down the process), `OutOfMemoryException` (catching it rarely helps; the allocation that follows your catch often re-throws), `AccessViolationException` (by default *not* delivered to managed `catch` blocks on .NET Core / .NET 5+ — it crashes the

process unless you opt into legacy delivery via the `legacyCorruptedStateExceptionsPolicy` setting), and `ThreadAbortException`. The last is a .NET Framework relic: on modern .NET, `Thread.Abort()` throws `PlatformNotSupportedException`, so `ThreadAbortException` is never raised — you list it for historical completeness, not because you'll encounter it in new code. Equally, `OperationCanceledException` should almost never be caught-and-swallowed because it is *not a failure* — more on that shortly.

A practical rule for library authors: derive your exceptions from a single library-specific base so callers can `catch` (`MyLibraryException`) to scope their handling, and never derive from `ApplicationException` (a vestige from .NET Framework that adds nothing and is officially discouraged).

```
// A library-rooted hierarchy: callers can catch the  
    base to scope handling,  
// or catch a leaf to act on a specific, actionable  
    condition.  
public class PaymentException : Exception  
{  
    public PaymentException(string message, Exception?  
        inner = null)  
        : base(message, inner) { }  
}  
  
public sealed class PaymentDeclinedException :  
    PaymentException  
{  
    // Actionable data on the exception – the caller  
    can branch on DeclineCode  
// without parsing a string message.  
    public required string DeclineCode { get; init; }  
    public bool IsRetriable => DeclineCode is  
        "issuer_unavailable" or "processing_error";  
  
    public PaymentDeclinedException(string message) :  
        base(message) { }  
}
```

The `required` member (C# 11) on `DeclineCode` forces every construction site to supply the actionable field, so you cannot accidentally throw a half-populated exception that the caller can't branch on.

Exception filters, rethrow, and the stack-trace trap

The single most damaging error-handling antipattern in C# is `throw ex;`. It resets the exception's stack trace to the rethrow point, erasing the original throw site — the one piece of information you actually needed.

```
try
{
    Process(order);
}
catch (PaymentException ex)
{
    Audit(ex);
    throw ex;    // BUG: stack trace now starts HERE,
                // original throw site lost.
}
```

The fix is a bare `throw;`, which preserves the original stack trace by rethrowing the *in-flight* exception object rather than re-initiating it:

```
catch (PaymentException ex)
{
    Audit(ex);
    throw;    // Correct: original throw site and
            // full stack preserved.
}
```

Exception filters (the `when` clause, C# 6) are strictly better than catch-and-throw for conditional handling, because a filter that evaluates to `false` never enters the catch block — the stack is *not unwound*, so a debugger breaks at the original throw site and the exception continues propagating with its trace intact. The canonical case is reacting to a specific HTTP status. `HttpRequestException.StatusCode` (a nullable `HttpStatusCode?`, added in .NET 5) makes this clean:


```

using System.Net;
using System.Net.Http;

async Task<string> FetchWithBackoffAsync(HttpClient
    http, Uri uri, CancellationToken ct)
{
    try
    {
        using var resp = await http.GetAsync(uri, ct);
        resp.EnsureSuccessStatusCode();
        return await
            resp.Content.ReadAsStringAsync(ct);
    }
    catch (HttpRequestException e) when (e.StatusCode
        == HttpStatusCode.TooManyRequests)
    {
        // Only THIS catch runs for 429s; any other
        status flows past untouched,
        // and because the filter gated entry, an
        unhandled 500 still shows its
        // true origin in the debugger.
        await Task.Delay(TimeSpan.FromSeconds(2), ct);
        throw; // let an outer retry policy decide;
        we just annotated/logged.
    }
}

```

Two subtleties worth internalizing. First, filters run **before** any stack unwinding, on top of the throwing frame — so a filter that itself throws will swallow that exception (the runtime treats a faulting filter as `false` and continues searching). Keep filter expressions pure, cheap, and side-effect-free; a filter is the wrong place to log if logging can throw. Second, a filter is the idiomatic way to log-and-rethrow *without* a catch: `catch (Exception e) when (Log(e))` where `Log` always returns `false`. The exception is logged at its origin and then continues propagating as if never caught — no stack reset, no `throw;` needed.

ExceptionDispatchInfo: replaying an exception across a boundary

Sometimes you catch an exception on one thread, stash it, and need to rethrow it later — or on a *different* thread — while preserving the original stack trace. A bare `throw;` only works in-

side the `catch` that caught it. The tool for “rethrow this captured exception elsewhere with its trace intact” is

`ExceptionDispatchInfo` (`System.Runtime.ExceptionServices`, since .NET 4.5). It captures the exception plus its stack and, on `Throw()`, *appends* the rethrow site rather than overwriting the origin.

```

using System.Runtime.ExceptionServices;
using System.Threading;

// A worker thread does fallible work; we marshal any
// failure back to the
// orchestrating thread and replay it there with the
// ORIGINAL stack preserved.
static Exception RunOnWorkerAndReplay()
{
    ExceptionDispatchInfo? captured = null;
    using var done = new ManualResetEventSlim();

    var worker = new Thread(() =>
    {
        try
        {
            DeepCall(); // throws somewhere down a
                        // real call chain
        }
        catch (Exception ex)
        {
            // Capture, don't rethrow on this thread.
            captured =
                ExceptionDispatchInfo.Capture(ex);
        }
        finally
        {
            done.Set();
        }
    });

    worker.Start();
    done.Wait();

    try
    {
        captured?.Throw(); // Replays with origin
                           // stack + "--- End of stack trace
                           // from previous location
                           // ---" marker, then THIS site.
        return null!;
    }
    catch (Exception replayed)
    {
        return replayed; // Stack shows DeepCall's
                        // frames, not just here.
    }

    static void DeepCall() =>

```

```
        throw new InvalidOperationException("failed  
        deep in the worker");  
    }
```

`ExceptionDispatchInfo.Capture(ex).Throw()` is what `await` does under the hood when an awaited task faults — it is how a synchronous-looking `await` surfaces an exception that actually occurred on a thread-pool thread with the original trace intact. You rarely write it by hand, but when you build your own task-like types, channel consumers, or actor mailboxes that ferry exceptions between threads, this is the correct primitive. Two related statics let you stamp a stack onto a *freshly-created* (never-thrown) exception before throwing it, which is useful in custom marshaling layers:

`ExceptionDispatchInfo.SetCurrentStackTrace(ex)` (added in .NET 5) captures the current call stack into `ex`, and `ExceptionDispatchInfo.SetRemoteStackTrace(ex, traceString)` (added in .NET 6) sets `StackTrace` from an arbitrary string — handy when reconstituting an exception that crossed a process or serialization boundary. Both return the same exception so you can `throw` `ExceptionDispatchInfo.SetCurrentStackTrace(new MyException());`.

Validation: guard clauses, throw helpers, and required members

Validation of *arguments* (contract enforcement) is distinct from validation of *user input* (expected, recoverable). Arguments get guard clauses that throw; user input gets a `Result` or a validation-error list. The BCL throw-helpers, fleshed out across .NET 6–8, make argument guards a one-liner and — importantly — keep the throwing slow path out of the caller's inlined fast path.

`ArgumentNullException.ThrowIfNull` arrived in .NET 6; `ArgumentException.ThrowIfNullOrEmpty` / `ThrowIfNullOrEmptySpace` in .NET 7/8; and the `ArgumentOutOfRangeException` family — `ThrowIfNegative`, `ThrowIfNegativeOrZero`, `ThrowIfZero`, `ThrowIfGreaterThan`, `ThrowIfLessThan`,

`ThrowIfEqual`, and friends — in .NET 8. They use `[CallerArgumentExpression]` so the parameter name is captured automatically; you never hand-type a `nameof`.

```
using System.Net.Http;

public sealed class TransferService(HttpClient http)
{
    // Public surface: every argument guarded before any work begins.
    public async Task<Receipt> TransferAsync(
        Account from, Account to, decimal amount, int
        maxRetries,
        CancellationToken ct = default)
    {
        ArgumentNullException.ThrowIfNull(from);
        ArgumentNullException.ThrowIfNull(to);
        // ThrowIfNegativeOrZero: amount must be strictly positive.
        ArgumentOutOfRangeException.ThrowIfNegativeOrZero(amount);
        // paramName captured automatically as "maxRetries".
        ArgumentOutOfRangeException.ThrowIfNegative(maxRetries);
        ArgumentOutOfRangeException.ThrowIfGreaterThan(maxRetries, 10);

        // ... perform the transfer; arguments are now known-valid.
        return await ExecuteAsync(from, to, amount, ct);
    }

    private Task<Receipt> ExecuteAsync(Account a,
        Account b, decimal amt, CancellationToken ct)
        => Task.FromResult(new Receipt(Guid.NewGuid(), amt));
}

public sealed record Account(string Iban);
public sealed record Receipt(Guid Id, decimal Amount);
```

Why the helpers beat hand-rolled `if (x is null) throw new...`: the throw lives behind a non-inlined method call, so the JIT keeps your method body small and the common (valid) path

stays in the instruction cache. The throw-helper analyzer family **CA1510–CA1513** flags hand-written equivalents and offers a fix — CA1510 for `ArgumentNullException` , CA1511 for `ArgumentException` , CA1512 for `ArgumentOutOfRangeException` , and CA1513 for `ObjectDisposedException` .

For *object* validation, prefer the type system over runtime checks. The `required` modifier (C# 11) plus `init` accessors guarantees a property is set at construction, eliminating a whole class of “forgot to set X” `NullReferenceException` s without any runtime guard. And in C# 14, the **field contextual keyword** (new in C# 14 / .NET 10) lets you fold validation directly into an auto-property’s setter without declaring an explicit backing field — and crucially, nothing else in the class can bypass that setter, because `field` is only visible inside the property:

```

public sealed class Quote
{
    // C# 14: 'field' is the compiler-synthesized
    // backing field, scoped to the
    // property. Here one accessor stays auto-
    // implemented (the bare 'get;') while
    // the other has a custom body – mixing auto and
    // full accessors is exactly
    // what C# 14 newly allows. Validation lives in
    // the setter; the rest of the
    // class cannot write the backing field directly
    // and circumvent the check.
    public decimal Price
    {
        get;
        set => field = value >= 0
            ? value
            : throw new
                ArgumentOutOfRangeException(nameof(value),
                    "Price must be non-negative.");
    }

    // Both accessors are custom here, and the getter
    // reads 'field' directly –
    // also enabled by the C# 14 'field' keyword (no
    // explicit backing field).
    public string Symbol
    {
        get => field;
        set => field =
            string.IsNullOrEmpty(value)
                ? throw new ArgumentException("Symbol
                    required.", nameof(value))
                : value.ToUpperInvariant();
    }
}

```

Before C# 14 this required a hand-written `private decimal _price;` that any method in the class could mutate around the validation. The `field` keyword closes that hole.

Cancellation is not failure

`OperationCanceledException` (and its subtype `TaskCanceledException`) is the one exception type that crosses the type system as “control flow,” not “error.” When a `CancellationToken` is signaled, cooperative APIs throw it — and your handling must distinguish it from a genuine fault, be-

cause logging a cancellation as an error pollutes telemetry, and *retrying* a cancellation is actively wrong.

```
async Task<Report?>
    TryBuildReportAsync(CancellationTokentoken ct)
{
    try
    {
        return await BuildAsync(ct);
    }
    // Filter on the token so we only treat OUR
    // cancellation as cancellation,
    // not a timeout from some inner library that
    // reused the type.
    catch (OperationCanceledException) when
        (ct.IsCancellationRequested)
    {
        // Expected shutdown/cancel – do NOT log as
        // error, do NOT retry.
        return null;
    }
    catch (Exception ex)
    {
        // Genuine failure path: log, maybe retry,
        // surface.
        Logger.LogError(ex, "Report build failed");
        throw;
    }
}
```

The `when (ct.IsCancellationRequested)` filter matters: an inner `HttpClient` timeout *also* surfaces as `TaskCanceledException` even when *your* token wasn't signaled. Without the filter you'd mistake a network timeout (a transient failure you might retry) for a deliberate cancellation (which you must not). You can disambiguate two further ways: `OperationCanceledException.CancellationToken` (present since .NET Framework) tells you *which* token cancelled the operation, so you can compare it against your own; and since .NET 6, an `HttpClient` request that times out (as opposed to being cancelled by your token) throws a `TaskCanceledException` whose `InnerException` is a `TimeoutException` — checking for that lets you tell “the server was too slow” apart from “the caller walked away.”

A second rule: never catch `Exception` so broadly that it absorbs `OperationCanceledException` silently. The pattern above handles cancellation *first*, explicitly, before the general catch — order matters because the first matching `catch` wins.

Result/Option modeling versus exceptions

Exceptions are expensive on the throw path: constructing one walks the stack to capture the trace, and the throw itself involves a runtime search for a handler. On a hot path that “fails” routinely — parsing, lookups, validation of untrusted input — that cost is real (hundreds of nanoseconds to microseconds per throw, dwarfing the actual work). For *expected* failure, model the outcome as a value. The BCL already does this with the `Try*` pattern (`int.TryParse` , `Dictionary.TryGetValue`) and with `ValueTask` /nullable returns.

For richer error data, a `Result<T, TError>` as a `readonly record struct` gives you zero-allocation, value-typed success/failure with exhaustive pattern matching:

```

public readonly record struct Result<T, TError>
{
    private readonly T? _value;
    private readonly TError? _error;
    public bool IsSuccess { get; }

    private Result(T value) { _value = value; _error
        = default; IsSuccess = true; }
    private Result(TError e) { _error = e;
        _value = default; IsSuccess = false; }

    public static Result<T, TError> Ok(T value)
        => new(value);
    public static Result<T, TError> Fail(TError error)
        => new(error);

    // Implicit conversions make call sites read
    naturally: `return value;`.
    // CAVEAT: if T and TError are ever the same type
    these two operators collide
    // (CS0557, duplicate conversion) and you must
    drop the implicits and call
    // Ok/Fail explicitly. Keep the success and error
    types distinct.
    public static implicit operator Result<T, TError>
        (T value) => Ok(value);
    public static implicit operator Result<T, TError>
        (TError error) => Fail(error);

    // Exhaustive, allocation-free consumption.
    public TResult Match<TResult>(  
        Func<T, TResult> onOk,  
        Func<TError, TResult> onError)  
        => IsSuccess ? onOk(_value!) :  
            onError(_error!);

    // Safe deconstruction for pattern matching.
    public bool  
        TryGet([System.Diagnostics.CodeAnalysis.MaybeN  
            ullWhen(false)] out T value,  
  
        [System.Diagnostics.CodeAnalysis.MaybeNullWhen  
            (true)] out TError error)  
    {  
        value = _value;  
        error = _error;  
        return IsSuccess;  
    }
}

public enum OrderError { OutOfStock, PaymentDeclined,  
    InvalidAddress }

```

```

// Producer: a contract violation (qty <= 0) THROWS –
// that's a bug in the caller,
// not an expected outcome. The genuinely expected
// failures flow back as values.
static Result<Receipt, OrderError> PlaceOrder(int sku,
int qty)
{
    ArgumentOutOfRangeException.ThrowIfNegativeOrZero(qty);
    if (!InStock(sku, qty)) return
        OrderError.OutOfStock;
    return new Receipt(Guid.NewGuid(), qty); //
        implicit -> Ok
}

// Consumer: pattern-matched, branch-per-error.
// NOTE the guard logic: TryGet returns IsSuccess, so
// on the FAILURE arm we test
// `is false` (the call still runs and assigns `err`,
// and the `is false` test is
// true precisely when we have an error). Writing
// `when r.TryGet(out _, out var err)`
// without the `is false` would be the success
// condition and the error arm would
// never run – a subtle, easy-to-miss bug.
static string Describe(int sku, int qty) =>
    PlaceOrder(sku, qty) switch
    {
        var r when r.TryGet(out var receipt, out _)
            => $"Confirmed {receipt.Id}",
        var r when r.TryGet(out _, out var err) is
            false
            => err switch
            {
                OrderError.OutOfStock =>
                    "Sorry, out of stock.",
                OrderError.PaymentDeclined =>
                    "Payment was declined.",
                OrderError.InvalidAddress =>
                    "Check your address.",
                _ =>
                    "Unknown error."
            },
        _ => "Unknown error."
    };

static bool InStock(int sku, int qty) => qty < 100;

```

The tradeoffs are real and you should weigh them deliberately. `Result` makes failure explicit in the signature and forces the

caller to handle it — but C# has no enforced exhaustiveness on `switch` and no monadic `?` propagation operator, so chaining many fallible steps gets verbose compared with F# or Rust. It also doesn't compose with `using` / `finally` cleanup the way exceptions do, and it can't carry a stack trace. **Use `Result` for domain-expected failures on warm/hot paths and at API boundaries where the failure is part of the contract; use exceptions for contract violations, truly exceptional I/O faults, and anywhere the call stack three frames up has no business knowing about the failure.** Mixing both deliberately — `Result` for the 99% expected case, exceptions for the 1% catastrophic — is idiomatic, not a smell.

Resilience pipelines: retry, timeout, circuit breaker

Transient faults — a 429, a dropped connection, a slow dependency — are not bugs and not domain errors; they are the weather. `Microsoft.Extensions.Resilience` and `Microsoft.Extensions.Http.Resilience` (built on Polly v8, current package versions track the `10.x` line for .NET 10) give you composable strategies. The cleanest integration for HTTP is to attach a resilience handler to a typed/named `HttpClient` so every outgoing call is wrapped, with zero changes to call sites.

The fastest path is `AddStandardResilienceHandler()`, which layers — outermost to innermost — a rate limiter (1000 permits, no queue), a total-request timeout (30s), a retry with exponential backoff and jitter (3 retry attempts, i.e. up to 4 tries total), a circuit breaker (30s sampling window, 0.1 / 10% failure ratio, minimum throughput 100, 5s break), and a per-attempt timeout (10s). When you need explicit control, build the pipeline yourself. Strategy **order matters**: the first added is the outermost. Retry must wrap timeout (so each attempt gets its own timeout), and the circuit breaker typically sits inside retry but outside the per-try timeout so that timeouts count toward tripping the breaker.

```

using System.Net;
using System.Net.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Http.Resilience;
using Polly;
using Polly.CircuitBreaker;
using Polly.Retry;
using Polly.Timeout;

var services = new ServiceCollection();
services.AddLogging();

services.AddHttpClient("catalog", c => c.BaseAddress =
    new Uri("https://api.example.com"))
    .AddResilienceHandler("catalog-pipeline", builder
        =>
    {
        // 1. OUTERMOST: total timeout across all
        //    retries (defense against a
        //    pathological retry storm running
        //    unbounded).
        builder.AddTimeout(TimeSpan.FromSeconds(30));

        // 2. Retry transient faults with exponential
        //    backoff + jitter.
        builder.AddRetry(new HttpRetryStrategyOptions
        {
            MaxRetryAttempts = 3,
            BackoffType =
            DelayBackoffType.Exponential,
            UseJitter = true,
            // avoid thundering herd
            Delay =
            TimeSpan.FromMilliseconds(200),
            // ShouldHandle defaults to transient HTTP
            // errors (5xx, 408, 429,
            // HttpRequestException,
            // TimeoutRejectedException). Customize if
            // needed:
            ShouldHandle = static args =>
            ValueTask.FromResult(
                args.Outcome switch
                {
                    { Result.StatusCode:
                    HttpStatusCode.TooManyRequests } => true,
                    { Result.StatusCode: >=
                    HttpStatusCode.InternalServerError } => true,
                    { Exception: HttpRequestException }
                    => true,
                    { Exception:
                    TimeoutRejectedException }
                    => true,
                }
            )
        })
    }
    );

```

```

        _ => false,
    });

    // 3. Circuit breaker: stop hammering a down
    //      dependency. Sits inside
    //      retry so a tripped breaker fails fast
    //      and short-circuits retries.
    builder.AddCircuitBreaker(new
    HttpCircuitBreakerStrategyOptions
    {
        FailureRatio      = 0.5,
        // open if >50% of...
        SamplingDuration =
        TimeSpan.FromSeconds(10),
        MinimumThroughput = 8,
        // ...at least 8 calls fail
        BreakDuration     =
        TimeSpan.FromSeconds(15)
    });

    // 4. INNERMOST: per-attempt timeout (each try
    //      gets a fresh budget).
    builder.AddTimeout(TimeSpan.FromSeconds(5));
});

var provider = services.BuildServiceProvider();
var factory =
    provider.GetRequiredService<IHttpClientFactory>
    >();

async Task<string> GetCatalogAsync(CancellationToken
    ct)
{
    var http = factory.CreateClient("catalog");
    // Every call now flows through timeout -> retry -
    // > breaker -> timeout.
    // A 429 is retried with backoff; a flapping
    //      dependency trips the breaker;
    // a hung socket is cut at 5s per try and 30s
    //      overall.
    using var resp = await http.GetAsync("/v1/items",
        ct);
    resp.EnsureSuccessStatusCode();
    return await resp.Content.ReadAsStringAsync(ct);
}

```

Operational notes that separate a working pipeline from a production one. When the breaker is open, calls throw `BrokenCircuitException` — handle it as “fail fast” (serve cached data, degrade gracefully), not as a generic 500. A per-at-

tempt timeout surfaces as `TimeoutRejectedException`, which is what your retry's `ShouldHandle` must treat as transient. Critically, your `CancellationToken` still flows through the whole pipeline: a *caller* cancellation short-circuits everything and surfaces `OperationCanceledException` — which, per the cancellation section, you must not retry. And do not stack a resilience handler on top of `AddStandardResilienceHandler` — you'll double-retry. Pick one.

Logging exceptions without lying to yourself

Two failure modes dominate exception logging. The first is **double-logging**: every layer catches, logs, and rethrows, so one failure produces five stack traces in your aggregator and your error-rate dashboards triple-count. The discipline: **log where you handle, rethrow where you don't**. A layer that cannot recover should let the exception propagate untouched (or annotate via a `when`-filter that returns `false`) and let the outermost handler — usually framework middleware — do the single authoritative log.

The second is **stringly-typed logs**:

`logger.LogError($"Failed for {orderId}: {ex}")` defeats structured logging. Pass the exception as the first argument and use a message template with named holes, so the logging pipeline captures `OrderId` as a queryable field and the exception as a first-class object with its full trace:

```

using Microsoft.Extensions.Logging;

try
{
    await ProcessAsync(order, ct);
}
catch (PaymentDeclinedException ex)
{
    // Structured: OrderId and DeclineCode become
    // indexed fields, ex is logged
    // once with its full stack. No string
    // interpolation of the message holes.
    Logger.LogWarning(ex, "Payment declined for order
        {OrderId} with code {DeclineCode}",
        order.Id, ex.DeclineCode);

    // We HANDLED it (declines are a known business
    // outcome) -> do not rethrow.
    await NotifyCustomerAsync(order, ct);
}

```

Use the right level: a handled, expected decline is `LogWarning` ; an unhandled fault you're about to rethrow is `LogError` at the boundary only; cancellation is `LogDebug` or not logged at all. For the absolute hot path, the source-generated `[LoggerMessage]` attribute compiles the template once and avoids boxing and allocation entirely — prefer it for high-frequency logging sites.

A worked example: tying it together

The following service combines guard clauses, a resilience pipeline, cancellation discipline, an exception filter, `Result` modeling for the expected-failure case, and single-point structured logging — the shape of a real outbound integration.


```

using System.Net;
using System.Net.Http;
using System.Net.Http.Json;
using Microsoft.Extensions.Logging;
using Polly.CircuitBreaker;

public sealed class PricingClient(HttpClient http,
    ILogger<PricingClient> logger)
{
    // Returns a Result for the EXPECTED outcomes
        (found / not found / unavailable);
    // throws only for programmer error (bad args)
        which should never reach prod.
    public async Task<Result<decimal, PriceError>>
        GetPriceAsync(
            string symbol, CancellationToken ct = default)
    {
        // Contract guards: violations are bugs, so
            they throw.

        ArgumentException.ThrowIfNullOrWhiteSpace(symbol);

        try
        {
            // http already has the resilience handler
                attached at registration.
            using var resp = await
                http.GetAsync($"v1/price/{symbol}", ct);

            if (resp.StatusCode ==
                HttpStatusCode.NotFound)
                return PriceError.NotFound;
            // expected -> value

            resp.EnsureSuccessStatusCode();
            var dto = await
                resp.Content.ReadFromJsonAsync<PriceDto>(ct);
            // NOTE: can't write `dto?.Price ??
                PriceError.NotFound` - decimal and
                // PriceError have no common type for ??,
                so the operands don't unify.
            // Branch explicitly; each branch then
                implicitly converts to Result.
            if (dto is null)
                return PriceError.NotFound;
            // expected -> value
            return dto.Price;
            // implicit -> Ok
        }
        // Caller cancellation is control flow, not
            failure: surface it, don't map.
    }
}

```

```

        catch (OperationCanceledException) when
            (ct.IsCancellationRequested)
        {
            throw;
        }
        // Breaker open == dependency is down: degrade
        // gracefully, log once at warn.
        catch (BrokenCircuitException ex)
        {
            logger.LogWarning(ex, "Pricing circuit
            open for {Symbol}", symbol);
            return PriceError.Unavailable;
        }
        // A 429 that survived the retry policy:
        // expected-ish, distinct outcome.
        catch (HttpRequestException ex) when
            (ex.StatusCode ==
            HttpStatusCode.TooManyRequests)
        {
            logger.LogWarning(ex, "Pricing rate-
            limited for {Symbol}", symbol);
            return PriceError.Unavailable;
        }
    }
}

public enum PriceError { NotFound, Unavailable }
public sealed record PriceDto(decimal Price);

// Caller consumes the Result with exhaustive matching
// – no try/catch needed
// for the expected outcomes, because they are modeled
// as values.
static async Task ShowPriceAsync(PricingClient client,
    string symbol, CancellationToken ct)
{
    var result = await client.GetPriceAsync(symbol,
        ct);
    var line = result.Match(
        onOk: price => $"{symbol}: {price:C}",
        onError: err => err switch
        {
            PriceError.NotFound => $"{symbol}: no
            price on file",
            PriceError.Unavailable => $"{symbol}:
            pricing temporarily unavailable",
            _ => $"{symbol}:
            error"
        });
    Console.WriteLine(line);
}

```

Notice what each construct buys you: the guard clause keeps a bug from silently producing garbage; the resilience handler absorbs the weather before your code ever sees it; the cancellation filter keeps shutdown out of your error logs; the breaker filter degrades instead of cascading; and the `Result` lets the *caller* handle “not found” without a `try/catch`, reserving exceptions for the genuinely exceptional.

Performance and the pitfalls, restated

Don't model control flow with exceptions on hot paths.

A throw captures a stack trace and triggers a handler search — easily $100\times$ – $1000\times$ the cost of a `bool` return. Parsing user input, dictionary misses, and validation of untrusted data should use `Try*` / `Result`, not `throw`. Reserve throwing for the genuinely rare.

Never `throw ex`; — always bare `throw`; inside a catch, and prefer a `when` filter when you only need to *inspect* before deciding to handle, since a false filter never unwinds the stack and never resets the trace. Use

`ExceptionDispatchInfo.Capture().Throw()` only when you must rethrow *outside* the original catch or on another thread.

Don't catch `Exception` broadly and swallow.

A blanket `catch (Exception)` will absorb `OperationCanceledException` (breaking cancellation), and although `StackOverflowException` can't be caught and `OutOfMemoryException` rarely should be, a broad catch still lets you accidentally continue execution in a corrupted state. Catch the narrowest type that you can actually act on; let everything else propagate to a single boundary handler that logs once.

Mind the GC and JIT cost of the happy path, not just the sad one.

The throw-helpers (`ArgumentNullException.ThrowIfNull` et al.) exist precisely so the *non-throwing* path stays inlinable and allocation-free; hand-rolled `if/throw` blocks can pessimize inlining of the whole method. And in resilience pipelines, exponential backoff *with jitter* (`UseJitter = true`) is not optional polish — without it,

every client retries in lockstep and converts a blip into a self-inflicted DDoS the instant the dependency recovers.

28 · Modern API and Library Design

A public API is a contract you can never fully retract: every signature you ship constrains every release that follows. This chapter is about designing that surface deliberately — making the call site discoverable and honest, choosing span-first shapes without abandoning allocating callers, evolving interfaces without breaking implementers, and using C# 13/14 features

(`[OverloadResolutionPriority]` , extension members, implicit span conversions, user-defined compound operators) where they earn their keep — and recognizing the cases where the safe, boring choice is the right one.

Designing for the call site

The single most useful discipline in library design is to write the *consuming* code first. The signature that reads cleanly at the call site is almost always the one worth implementing, even when it costs you internally. Three properties dominate: discoverability (the right member surfaces in IntelliSense at the right time), nullability honesty (annotations match runtime behavior), and overload sanity (the compiler picks the overload you intended, and the wrong one isn't even a candidate).

Discoverability is mostly about *scoping members to where they apply*. A method that only makes sense after `Build()` should live on the built type, not the builder. A static factory belongs on the type it produces. Fluent chains read well precisely because each return type narrows what you can do next. The enemy is the “god type” exposing forty members of which six are valid in any given state.

Nullability honesty is non-negotiable for a library, because consumers compile against your annotations and the compiler enforces them transitively. If a method can return `null` , annotate it `T?` even if it “almost never” does. The reverse failure is worse: annotating a parameter non-nullable when you actually tolerate `null` , then throwing `ArgumentNullException` — the compiler

told the caller it was safe, so the throw is a surprise. Annotate truthfully and validate at the boundary:

```
public sealed class UriRouteTable
{
    // Honest: returns null when no route matches.
    // Callers must handle it,
    // and the compiler will force them to.
    public RouteEntry? TryResolve(ReadOnlySpan<char>
        path) { /* ... */ return null; }

    // Honest: 'pattern' is required. Validate even
    // though the annotation
    // already nudges callers, because reflection/old
    // assemblies bypass it.
    public void Add(string pattern, RouteEntry entry)
    {
        ArgumentNullException.ThrowIfNull(pattern);
        // .NET 6+; concise guard
        ArgumentNullException.ThrowIfNull(entry);
        // ...
    }
}

public readonly record struct RouteEntry(string
    Handler, int Order);
```

`ArgumentNullException.ThrowIfNull` (introduced in .NET 6) captures the argument name via `[CallerArgumentExpression]`, so you never pass a `nameof`. Use it at every public entry point that declares a non-nullable reference parameter; the annotation is a *promise to compiled-against-you callers*, and the guard is the *enforcement for everyone else* (reflection, dynamic, downlevel assemblies, F#).

Span-first APIs, with allocating overloads

Modern .NET BCL design is span-first: the fast, allocation-free core takes a `Span<T>` / `ReadOnlySpan<T>` and the caller owns the buffer. But you rarely delete the allocating, convenient overload — you offer both, and have the convenient one delegate to the span core. This is the `Format` / `TryFormat` pattern that runs through the BCL (`int.TryFormat`, `DateTime.TryFormat`, `Guid.TryFormat`).

Here is a library temperature formatter exposing a span-based core and a string-returning convenience overload over the same logic:

```

using System;
using System.Buffers;
using System.Globalization;

public readonly record struct Temperature(double
    Celsius)
{
    // Span-first core: caller owns the buffer, zero
    allocations on success.
    // Returns false (not throws) when the destination
    is too small – the
    // standard ITryFormat contract.
    public bool TryFormat(Span<char> destination, out
        int charsWritten,
        ReadOnlySpan<char> format =
        default,
        IFormatProvider? provider =
        null)
    {
        provider ??= CultureInfo.CurrentCulture;
        // Compose "<value> °C" by writing the number
        then the suffix.
        if (!Celsius.TryFormat(destination, out int n,
            format, provider))
        {
            charsWritten = 0;
            return false;
        }
        ReadOnlySpan<char> suffix = " °C"; // " °C"
        if (destination.Length - n < suffix.Length)
        {
            charsWritten = 0;
            return false;
        }
        suffix.CopyTo(destination[n..]);
        charsWritten = n + suffix.Length;
        return true;
    }

    // Allocating convenience overload – delegates to
    the span core.
    // One allocation on the success path (the result
    string); the work buffer
    // is on the stack.
    public string Format(ReadOnlySpan<char> format =
        default,
        IFormatProvider? provider =
        null)
    {
        Span<char> buffer = stackalloc char[32]; //
        ample for a double + suffix
    }
}

```



```

        if (TryFormat(buffer, out int written, format,
            provider))
            return new string(buffer[..written]);

        // Rare slow path: the stack buffer was too
        // small. Rent from the pool
        // (no Gen0 allocation for the work buffer)
        // and always return it.
        char[] rented =
            ArrayPool<char>.Shared.Rent(256);
        try
        {
            TryFormat(rented, out written, format,
                provider);
            return new string(rented, 0, written);
        }
        finally
        {
            ArrayPool<char>.Shared.Return(rented);
        }
    }
}

```

Usage and the tradeoff:

```

var t = new Temperature(21.5);

// Hot path: no allocation, caller-owned buffer.
Span<char> buf = stackalloc char[32];
if (t.TryFormat(buf, out int len, format: "F1"))
    Console.Out.Write(buf[..len]);           // "21.5
                                             °C"

// Convenience path: one string allocation, identical
// formatting logic.
string s = t.Format("F1");                  // "21.5
                                             °C"

```

Design rules this encodes: the span overload returns `bool` and never throws on capacity (the `Try` contract); the allocating overload is *thin*, never duplicating formatting logic; and the parameter order matches the BCL (`destination, out charsWritten, format, provider`) so the type drops into generic formatting machinery. Note `ReadOnlySpan<char> format = default` rather than `string?` — as of C# 14 a `string` literal implicitly converts to `ReadOnlySpan<char>`, so

callers still write `"F1"` and you avoid materializing a `string` you only read.

The performance picture is worth making concrete. The `TryFormat` path uses `stackalloc char[32]`, which the JIT lowers to a stack-frame reservation — no GC allocation, and the buffer dies when the method returns. The C# spec mandates that `stackalloc` zero-initializes the allocated memory (64 bytes here), so there is a small fixed zeroing cost; you can elide it with `[SkipLocalsInit]` on the method when you fully overwrite the buffer, but that is an unsafe-ish micro-optimization to reach for only after measuring. The `Format` path adds exactly one heap allocation: the result `string (new string(buffer[..written])` copies the written span into a freshly allocated, immutable string). Compare this with a naive `Celsius.ToString("F1") + " °C"`, which allocates *twice* (the intermediate `"21.5"` string and the concatenation result) — so even the convenience overload is a measurable win, and the span overload removes the heap entirely. For a formatter called millions of times in a logging or serialization hot loop, that is the difference between flat and sawtooth Geno pressure.

When *not* to go span-first: if the value's natural lifetime escapes the call (you store it, box it, hand it to an `async` continuation that outlives the buffer), a span parameter is the wrong tool — spans are `ref struct`s and cannot cross `await`, be captured by a lambda, or live on the heap. Offer the span overload for *transformation*, the allocating one for *retention*. And don't over-stackalloc: a buffer sized from untrusted input can blow the stack, so cap the stack buffer and fall back to `ArrayPool<char>.Shared.Rent` (not a fresh `new char[]`) for the cold path in genuinely hot code.

First-class span conversions and steering with

`[OverloadResolutionPriority]`

C# 14 (.NET 10) added first-class implicit conversions among `T[]`, `Span<T>`, and `ReadOnlySpan<T>`, and `string` → `ReadOnlySpan<char>`. This lets you collapse overload sets and

write one `ReadOnlySpan<T>` method that serves both spans and arrays. But the conversions also make span overloads *applicable* in more places, which has two consequences a library author must reason about precisely.

It helps to be exact about what the new rules do and do *not* do. When a `T[]` argument is passed and the candidate set holds *both* a `T[]` overload and a `ReadOnlySpan<T>` overload, the call is **not** ambiguous: the array argument is an identity match to the `T[]` parameter, and an identity conversion is strictly better than an implicit span conversion under the “better conversion from expression” rule. The `T[]` overload still wins. Where the span change actually bites is the *new applicability* case — most visibly extension methods. In C# 13 a `ReadOnlySpan<T>` extension method was simply not applicable to a `T[]` receiver; in C# 14 it is, so a call that previously bound to (say) `Enumerable.Contains` can now bind to `MemoryExtensions.Contains` instead. (That specific rebinding is the .NET 10 breaking change the BCL documents for Expression-tree interpretation, where the span method throws at compile-of-the-tree time.)

The genuine *ambiguity* case — and the one `[OverloadResolutionPriority]` (introduced in C# 13) was designed for — is a single argument that is implicitly convertible to *multiple* parameter types with no clear winner, e.g. a type exposing implicit conversions to both `string` and `string[]` against `params string[]` vs `params ReadOnlySpan<string>` overloads. The attribute also serves a second, distinct purpose: deliberately *steering* binding to a preferred overload even when another candidate would otherwise win, so you can make a legacy overload vestigial without deleting it.

Consider a library that shipped an array-based hashing API, then added a faster span core and wants *new* compilations to prefer it even when the caller still passes an array:

```

using System.Runtime.CompilerServices;

public static class ContentHasher
{
    // Legacy overload – shipped in v1, callers
    // compiled against it.
    // Without the attribute below,
    // Hash(someByteArray) binds HERE (identity
    // match).
    public static int Hash(byte[] data) =>
        HashCore(data);

    // v2 fast path. The priority attribute steers a
    // byte[] argument to THIS
    // overload even though byte[] is an identity
    // match for the array overload –
    // priority pruning removes the lower-priority
    // candidate before betterness runs.
    [OverloadResolutionPriority(1)]           // C#
    13; higher wins
    public static int Hash(ReadOnlySpan<byte> data) =>
        HashCore(data);

    private static int HashCore(ReadOnlySpan<byte>
        data)
    {
        var hc = new HashCode();
        hc.AddBytes(data);
        return hc.ToHashCode();
    }
}

```

```

byte[] bytes = [1, 2, 3, 4];
int h = ContentHasher.Hash(bytes);    // binds to the
    ReadOnlySpan<byte> overload

```

How this works in the spec: priority pruning is applied *before* the betterness rules. The candidate set is grouped by declaring type; within each group every candidate below the highest priority found is removed, and only then does normal “better function member” selection run. So `[OverloadResolutionPriority(1)]` on the span overload removes the array overload from the group entirely — the array overload never gets to win on its identity conversion. The legacy `byte[]` overload remains reachable via a delegate conversion or reflection, but is no longer the default winner. Two caveats the language team flags: the attribute is

silently ignored under `LangVersion < 13` (a consumer pinned to an older language version sees the un-steered binding, which can differ from yours); and priority groups *by declaring type*, so a base type cannot out-prioritize a derived type’s applicable member. Reserve it for genuine “two good overloads, prefer the fast one” or “resolve a real conversion ambiguity” cases — not as a band-aid for an overload set that is confusing to begin with.

Default interface members for safe evolution

Adding a member to a shipped `public interface` is normally a breaking change: every implementer fails to compile, and pre-existing assemblies throw `TypeLoadException` at runtime because they don’t supply the new member. Default interface members (DIMs), introduced in C# 8, let you add a member *with a body*, so existing implementers inherit a working default and only override when they want to.

Suppose v1 shipped this:

```
public interface IEventSink
{
    void Write(string category, string message);
}
```

In v2 you want batch writes. Adding `void WriteBatch(...)` as an abstract member breaks everyone. A DIM does not:

```

public interface IEventSink
{
    void Write(string category, string message);

    // Added in v2. Existing implementers keep
    // compiling and running:
    // they inherit this body, which loops over the v1
    // method.
    // High-throughput sinks can override for a true
    // batch path.
    void WriteBatch(ReadOnlySpan<(string Category,
        string Message)> events)
    {
        foreach (var (category, message) in events)
            Write(category, message);
    }
}

```

```

// v1 implementer – compiled before WriteBatch
// existed. Still valid.
public sealed class ConsoleSink : IEventSink
{
    public void Write(string category, string message)
        => Console.WriteLine($"[{category}]
        {message}");
    // Inherits the default WriteBatch – no recompile
    // required.
}

// v2 implementer opts into a real batch path.
public sealed class BufferedSink : IEventSink
{
    public void Write(string category, string message)
        { /* ... */ }

    // Element names must match the interface
    // declaration, or the compiler
    // warns (CS8141) that the tuple names differ from
    // the implemented member.
    public void WriteBatch(ReadOnlySpan<(string
        Category, string Message)> events)
    {
        // single lock acquisition, one flush, etc.
    }
}

```

Critical subtlety: a DIM is only callable *through the interface*, not through the implementing class, unless re-declared. `new`

`ConsoleSink().WriteBatch(...)` does **not** compile; `((IEventSink)sink).WriteBatch(...)` does. Dispatch is virtual through the interface, so the override is honored regardless of the static type. DIMs are the right tool for *additive* interface evolution and for providing a safe default of an *optional* capability. They are the wrong tool for sharing state (interfaces still cannot hold instance fields) or as a substitute for an abstract base class when you actually want a class hierarchy. Note also that DIMs require runtime support — they are a .NET Core 3.0+/.NET 5+ feature and do not light up on .NET Framework.

Extension members to shape a fluent surface (C# 14)

C# 14 generalizes extension methods into **extension members**: inside a top-level non-generic `static class`, an `extension(ReceiverType receiver) { ... }` block can declare extension *properties*, *static* extension methods/properties, and extension *operators*, in addition to ordinary instance extension methods. This lets a library — or a consumer of one — add a fluent, discoverable surface over a core type *without adding members to the type itself*, keeping the core small and the conveniences in a separate, opt-in namespace.

The old `this`-parameter form and the new block form compile to identical IL and are both source- and binary-compatible; the block form simply unlocks member kinds (properties, operators, statics) that the `this` form never could.

Here a library owns a minimal `Money` core and ships an extensions class that layers a fluent surface on top:

```

using System.Globalization;

// Core type: deliberately tiny. Just the data and
// arithmetic invariant.
public readonly record struct Money(decimal Amount,
    string Currency);

// Fluent surface lives OUTSIDE the core, in its own
// static class.
// Consumers get these members via a using of this
// namespace.
public static class MoneyExtensions
{
    extension(Money money)
    {
        // Instance extension PROPERTY (C# 14) – reads
        // like a real member.
        public bool IsZero => money.Amount == 0m;

        // Instance extension METHOD.
        public Money Rounded(int decimals = 2)
            => money with { Amount =
            decimal.Round(money.Amount, decimals) };

        public string ToDisplay(IFormatProvider?
            provider = null)
            => money.Amount.ToString("N2", provider ??
            CultureInfo.CurrentCulture)
            + " " + money.Currency;
    }

    // Static extension members: no receiver name
    // needed when all members
    // are static. These appear as static members OF
    // Money to callers.
    extension(Money)
    {
        public static Money Zero(string currency) =>
            new(0m, currency);

        // Extension OPERATOR (C# 14): '+' over the
        // extended type, enforcing
        // the currency invariant without bloating the
        // core record.
        public static Money operator +(Money left,
            Money right)
        {
            if (left.Currency != right.Currency)
                throw new
                InvalidOperationException("Currency
                mismatch.");
        }
    }
}

```



```

        return left with { Amount = left.Amount +
            right.Amount };
    }
}

```

```

// A plain namespace `using` of the assembly that
// defines MoneyExtensions
// brings all of its extension members into scope – no
// `using static` needed.

Money a = Money.Zero("EUR");           // static
// extension method, reads as a factory
Money b = new(19.999m, "EUR");
Money sum = a + b;                       // extension
// operator
Console.WriteLine(sum.Rounded().ToDisplay()); //
// "20.00 EUR" (extension prop + methods)
Console.WriteLine(sum.IsZero);           // False –
// extension property

```

Why this matters for API design: the *core* `Money` carries only its invariant (amount + currency) and stays trivially serializable and equatable. The *ergonomics* — `IsZero`, `Zero`, display formatting, the currency-checked `+` — live in an extension class the consumer opts into with a `using`. You can ship several extension classes (display, persistence, math) and let consumers compose only what they need, and you can deprecate an extension surface without touching the core type's binary contract.

When *not* to reach for extension members: if a member is fundamental to the type's invariant or every consumer needs it, put it *on the type*. Extension members can't access private state, don't participate in the type's own `ToString`/equality, and won't be found by reflection over the type's members — so don't model core behavior as extensions. They shine for optional, call-site ergonomics and for adding a fluent layer over a type you *don't own* (a BCL or third-party type).

Contracts in the signature: `in`, `ref readonly`, and nullability

Parameter modifiers are part of your public contract and are encoded in metadata. `in` (C# 7.2) passes a readonly reference, avoiding a copy of a large struct; `ref readonly` (C# 12) is similar but makes the caller's intent explicit and warns when an rvalue is passed. For a library author the relevant facts are: switching a value parameter to `in` / `ref readonly` is a *source-compatible-ish* but *binary-breaking* change (the metadata signature changes), and `in` allows callers to omit the modifier at the call site while `ref readonly` encourages (and for non-variables, warns about) it.

```
public readonly struct Matrix4x4Like // 64 bytes –
    worth passing by reference
{
    // ... 16 floats ...

    // 'in': readonly reference, no defensive copy of
    // 64 bytes per call.
    // Caller writes Multiply(a, b) – no 'in' keyword
    // required at the call site.
    public static Matrix4x4Like Multiply(in
        Matrix4x4Like a, in Matrix4x4Like b)
        => default; // ...

    // 'ref readonly' (C# 12): same perf as 'in', but
    // the signature documents
    // that a real variable is expected; passing a
    // literal/temporary warns
    // (and 'ref readonly' callers may write the
    // modifier explicitly).
    public static float Determinant(ref readonly
        Matrix4x4Like m) => 0f; // ...
}
```

Two contract pitfalls deserve emphasis. First, `in` on a `readonly struct` avoids defensive copies; `in` on a *mutable* struct forces the compiler to make a hidden defensive copy on every member access through the parameter, quietly erasing the perf win you reached for — so pair `in` with `readonly struct`. Second — and this is the most insidious nullability break — **changing a parameter's nullable annotation breaks consumers**

compiled against the old one. Widening a return type's nullability (`string` → `string?`) breaks callers who relied on non-null; narrowing a parameter (`string?` → `string`) breaks callers who legitimately passed `null` . These don't change IL signatures (nullable annotations live in attributes), so they compile against the old binary but emit new warnings-as-errors and can crash. Treat the nullable annotations on your public surface as part of the versioned contract, not as advisory hints.

User-defined compound assignment operators (C# 14)

For library authors of large mutable value-or-buffer types — tensors, big integers, accumulators, geometry batches — C# 14 adds **user-defined instance compound assignment operators** (`+=` , `-=` , `*=` , `/=` , and friends) and **instance increment/decrement operators** (`++` , `--`). Before C# 14, `x += y` always expanded to `x = x + y` , which invokes the static binary `operator +` that *allocates and returns a new instance*. The new operators are declared as `void`-returning, non-`static` instance methods with the `operator` keyword, and they mutate the receiver in place — eliminating the temporary.

```

public sealed class Accumulator
{
    private double[] _buckets;
    public Accumulator(int size) => _buckets = new
        double[size];

    // Static binary operator: the classic, allocating
    // form. 'a += b' without
    // the instance operator below would call this and
    // allocate a new array.
    public static Accumulator operator +(Accumulator
        a, Accumulator b)
    {
        var result = new
            Accumulator(a._buckets.Length);
        for (int i = 0; i < a._buckets.Length; i++)
            result._buckets[i] = a._buckets[i] +
                b._buckets[i];
        return result;
    }

    // C# 14 instance compound assignment: void, non-
    // static, mutates in place.
    // 'acc += other' now binds HERE – zero
    // allocation, no array copy.
    public void operator +=(Accumulator other)
    {
        for (int i = 0; i < _buckets.Length; i++)
            _buckets[i] += other._buckets[i];
    }

    // C# 14 instance increment: void, non-static, no
    // parameters.
    public void operator ++()
    {
        for (int i = 0; i < _buckets.Length; i++)
            _buckets[i] += 1d;
    }
}

```

```

var acc = new Accumulator(1024);
var delta = new Accumulator(1024);
acc += delta;    // binds to the instance 'operator +=',
                 // in-place, no new array
acc++;           // binds to the instance 'operator ++',
                 // in-place

```

The binding rule (introduced in C# 14): when `x` in `x += y` is a *variable* and the language version is 14+, the compiler prefers a

user-defined instance `operator +=` if one is applicable; otherwise it falls back to the legacy `x = x + y` expansion. So shipping the instance `operator` is a transparent, source-compatible optimization for existing `+=` call sites — they keep compiling and simply stop allocating. The documented win on allocation-heavy types is dramatic (Microsoft’s own concert-simulation sample drops from 134 to 10 allocations, ~92%).

Three design cautions. The instance `operator +=` must not be `static` and must return `void` — that is the metadata signature (`op_AdditionAssignment`) the compiler looks for. There is no enforced consistency between `operator +` and `operator +=`, so you can accidentally make `a += b` and `a = a + b` behave differently; keep them semantically identical or you will astonish your users. And reach for these only on types where in-place mutation is genuinely safe and beneficial — immutable value types (the case where the old “allocate and return” was already correct and cheap) gain nothing and lose their immutability guarantee.

Options, builders, and configuration-friendly constructors

Three patterns dominate configurable libraries. A **constructor** is right when all dependencies are required and immutable — make them non-optional parameters so misuse is a compile error. The **options pattern** (an options object bound from `IConfiguration`) is right when configuration is data-shaped, hierarchical, and externally supplied. The **builder pattern** is right when construction is multi-step, has interdependent validation, or produces an immutable result from many optional inputs.

C# 12 primary constructors make the configuration-friendly constructor concise, and a `record`-based options class binds cleanly:

```

// Options pattern: a plain settings record, bound
// from configuration.
// Init-only so it's immutable after binding;
// nullable-honest defaults.
public sealed record HttpRetryOptions
{
    public int MaxAttempts { get; init; } = 3;
    public TimeSpan BaseDelay { get; init; } =
        TimeSpan.FromMilliseconds(200);
    public bool RetryOnTimeout { get; init; } = true;
}

// Builder: validates interdependent inputs and yields
// an immutable client.
public sealed class RetryPolicyBuilder
{
    private int _maxAttempts = 3;
    private TimeSpan _baseDelay =
        TimeSpan.FromMilliseconds(200);

    public RetryPolicyBuilder MaxAttempts(int value)
    {
        ArgumentOutOfRangeException.ThrowIfLessThan(va
            lue, 1); // .NET 8 guard
        _maxAttempts = value;
        return this;
    }

    public RetryPolicyBuilder BaseDelay(TimeSpan
        value)
    {
        ArgumentOutOfRangeException.ThrowIfLessThanOrE
            qual(value, TimeSpan.Zero);
        _baseDelay = value;
        return this;
    }

    // Build() returns the *built* type – narrowing
    // what the caller can do next.
    public RetryPolicy Build() => new(_maxAttempts,
        _baseDelay);
}

// Configuration-friendly constructor: required deps
// are non-optional.
public sealed class RetryPolicy(int maxAttempts,
    TimeSpan baseDelay)
{
    public int MaxAttempts { get; } = maxAttempts;

```

```
public TimeSpan BaseDelay { get; } = baseDelay;  
}
```

Two design rules: builders should validate *eagerly* in each setter (fail at the misconfiguring call, not at `Build()`), and the options object should never expose mutable collection state to consumers after binding — which is the next pitfall.

Worked example: a read-model registry tying it together

This registry shows span-first lookup, nullability honesty, read-only exposure of internal collections, a default interface member for evolution, and an extension member for ergonomics — the chapter's ideas in one compilable unit.

```

using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Diagnostics.CodeAnalysis;

public interface IReadModel
{
    string Key { get; }

    // DIM added in v2: a default validity check
    // existing models inherit.
    bool IsValid => !string.IsNullOrEmpty(Key);
}

public sealed class ReadModelRegistry
{
    private readonly Dictionary<string, IReadModel>
        _models = new();

    // PITFALL AVOIDED: expose a read-only VIEW, never
    // the live dictionary.
    // ReadOnlyDictionary wraps the same storage with
    // no copy, but blocks
    // mutation through the public surface.
    public IReadOnlyDictionary<string, IReadModel>
        Models { get; }

    public ReadModelRegistry() => Models = new
        ReadOnlyDictionary<string, IReadModel>
        (_models);

    public void Register(IReadModel model)
    {
        ArgumentNullException.ThrowIfNull(model);
        if (!model.IsValid) //
            // calls DIM or override
            throw new ArgumentException("Invalid read
            model.", nameof(model));
        _models[model.Key] = model;
    }

    // Nullability-honest lookup. 'out IReadModel?'
    // tells the caller the
    // out value is meaningful only when the method
    // returns true.
    public bool TryGet(ReadOnlySpan<char> key,
        [NotNullWhen(true)] out IReadModel? model)
    {
        // Dictionary<string, _> supports a span-based
        // lookup via its
        // alternate-comparer lookup (.NET 9+). For
        // brevity, materialize:
    }

```



```

        return _models.TryGetValue(key.ToString(), out
        model);
    }
}

// Extension member (C# 14): fluent ergonomics over
// the registry without
// bloating its core surface.
public static class RegistryExtensions
{
    extension(ReadModelRegistry registry)
    {
        public int ValidCount
        {
            get
            {
                int n = 0;
                foreach (var kvp in registry.Models)
                    if (kvp.Value.IsValid) n++;
                return n;
            }
        }
    }
}

```

```

var registry = new ReadModelRegistry();
registry.Register(new SnapshotModel("orders"));

if (registry.TryGetValue("orders", out var model))    //
    span-honest, NotNullWhen
    Console.WriteLine(model.Key);                    //
    "orders" – model is non-null here

Console.WriteLine(registry.ValidCount);              //
    extension property
// registry.Models["x"] = ...;                       //
    does NOT compile: read-only view

sealed file record SnapshotModel(string Key) :
    IReadModel;

```

`[NotNullWhen(true)]` (a nullable-flow attribute) is the crucial honesty touch: it tells the compiler that when `TryGet` returns `true`, `model` is guaranteed non-null, so the caller uses it without a null check and without a false warning. Combined with the `ReadOnlyDictionary` view, the registry is both safe and pleasant to consume.

Versioning, [Obsolete] , and experimental APIs

Distinguish **source compatibility** (consumer *source* still compiles against the new version) from **binary compatibility** (a consumer assembly compiled against the old version still *loads and runs* against the new one without recompiling). They diverge constantly. Adding an overload is source- and binary-compatible *unless* it changes overload resolution for existing call sites — then a *recompile* silently rebinds to the new overload (a source-level behavior break with no error). Adding an interface member is binary-breaking without a DIM. Reordering enum values, changing a `const` (it's baked into the consumer at compile time), tightening a parameter type, or changing a nullable annotation are all consumer-affecting in ways the table below summarizes:

Change	Source-compatible	Binary-compatible	Note
Add a method/overload	Usually	Yes	May rebind existing calls on recompile
Add abstract interface member	No	No	Use a DIM instead
Add default interface member	Yes	Yes	Requires .NET Core 3.0+ runtime
Widen return nullability <code>T → T?</code>	No	Yes (attrs)	Breaks callers' null-flow analysis
Change <code>const</code> value	Yes	No	Old value inlined into consumer
<code>value</code> param → <code>in / ref readonly</code>	Mostly	No	Signature changes in metadata

For deliberate deprecation, `[Obsolete]` (since C# 1) is the staged exit: warn first, then escalate to error in a later major version, then remove. The diagnostic-id overload lets consumers suppress *specific* obsoletions surgically:

```
public sealed class LegacyClient
{
    [Obsolete("Use SendAsync. Synchronous send is
        removed in v4.",
        error: false, // warning
        in v3...
        DiagnosticId = "MYLIB0007", // ...so
        callers can target-suppress
        UrlFormat =
        "https://docs.example.com/obsolete/{0}")]
    public void Send(string payload) { /* ... */ }

    public Task SendAsync(string payload) =>
        Task.CompletedTask;
}
```

For APIs that ship but aren't yet stable, `[Experimental]` (introduced in C# 12 / .NET 8, in

`System.Diagnostics.CodeAnalysis`) is stronger than `[Obsolete]`: using an experimental member is a *compile error by default*, and the consumer must explicitly opt in by suppressing the diagnostic. This makes “preview surface” honest — nobody depends on it accidentally.

```
using System.Diagnostics.CodeAnalysis;

[Experimental("LIB001")]
public sealed class VectorIndex
{
    // Entire type is experimental; touching it errors
    // with LIB001
    // unless the consumer opts in.
    public void Add(ReadOnlySpan<float> embedding) {
        /* ... */ }
}
```

```
// Consumer must opt in, surgically, to use the
    experimental API:
#pragma warning disable LIB001
var index = new VectorIndex();
#pragma warning restore LIB001
// - or project-wide via <NoWarn>LIB001</NoWarn> in
    the .csproj.
```

The diagnostic id must be alphanumeric (no dashes — they break `#pragma` parsing), which is why the convention is `LIB001`, not `LIB-001`. Apply `[Experimental]` to the whole type, a method, or a property; the error surfaces at every use site until opted in. When you stabilize the API, remove the attribute — that’s a non-breaking change for consumers who opted in, and it simply lets new consumers use it without the dance.

Pitfalls, restated as rules

Three failure modes from this chapter recur often enough to internalize as rules.

Don’t add an overload that silently changes resolution.

Overload sets and C# 14’s span conversions interact in non-obvious ways. A plain `Hash(byte[])` plus `Hash(ReadOnlySpan<byte>)` is *not* the danger — an array argument still binds to the array overload by identity. The danger is twofold: (a) adding a `ReadOnlySpan<T>` *extension method* now makes it applicable to array receivers that previously bound a different (e.g. `IEnumerable<T>`) extension — the .NET 10 behavioral break; and (b) reaching for `[OverloadResolutionPriority]` to steer binding, which *does* silently rebind `Hash(someArray)` from the array overload to the span overload on recompile, with no error and possibly different behavior or performance. The fix is to keep behavior identical across the overload set (so any rebind is harmless) and to apply `[OverloadResolutionPriority]` only deliberately, documenting the binding. Verify by recompiling a representative consumer and diffing the bound members (`/p:Features` IL diff, or a decompiler).

Don't expose mutable internal collections. Returning `List<T>` or `Dictionary<K,V>` directly hands callers a mutation handle into your internals; they will add and remove items, and your invariants silently rot. Expose `IReadOnlyList<T>` / `IReadOnlyDictionary<K,V>` backed by a no-copy wrapper (`ReadOnlyCollection<T>` , `ReadOnlyDictionary<K,V>`), or return an immutable snapshot. The wrapper is $O(1)$ and blocks mutation through the public surface — the registry example does exactly this.

Don't change a parameter's nullability annotation casually. It compiles against the old binary (annotations are attribute metadata, not IL signature), so you get no link-time error, but every consumer's null-flow analysis shifts — and the runtime behavior they relied on may now be wrong. Narrowing a parameter from `string?` to `string` breaks callers who passed `null` ; widening a return from `T` to `T?` breaks callers who dereferenced freely. Version nullability changes as you would any contract change: in a minor release if additive and honest, in a major release if it can break a compiling consumer.

The throughline: every one of these features — span overloads, `[OverloadResolutionPriority]` , DIMs, extension members, `[Experimental]` — is a tool for *controlling what the consumer sees and how it binds*. Reach for them when they make the contract clearer or safer to evolve, and leave them on the shelf when a plain method on a plain type says it better.

29 · Testing, Benchmarking, and Performance Validation

Correctness and performance are not separate disciplines — both are claims about behavior that you either measure or merely hope for. This chapter is about turning hope into evidence: testing the awkward shapes of modern C# (async streams, cancellation, `ref struct` Spans), proving allocation and codegen properties with BenchmarkDotNet, and wiring regression gates so a future commit can't silently undo your work.

Testing async, cancellation, and `IAsyncEnumerable`

The first rule of testing async code is to keep it async all the way down. `async Task` test methods are first-class in xUnit, NUnit, and MSTest; never bridge with `.Result` or `.Wait()`. Blocking on a `Task` from a synchronous test can deadlock when a captured `SynchronizationContext` tries to resume the continuation on a thread you're holding hostage — classic in UI and old ASP.NET contexts. Test runners install no sync context by default, so a blocked test usually *works* until someone adds a `[UISettings]` or a custom context, at which point it deadlocks intermittently. Treat any `.Result` in a test as a latent flake.

The harder target is `IAsyncEnumerable<T>`. You want to assert both the values produced *and* that cancellation propagates promptly. Here is a streaming source and a test that the stream throws `OperationCanceledException` when the token is cancelled mid-iteration:

```
using System.Runtime.CompilerServices;

public static class Telemetry
{
    // C# 13 (.NET 9) allowed `ref` locals and
    // `unsafe` contexts in iterators
    // and async methods (you still can't cross a
    // `yield`/`await` with the ref);
    // the [EnumeratorCancellation] plumbing is older
    // (C# 8) but essential.
    public static async IEnumerable<int>
        ReadingsAsync(
            [EnumeratorCancellation] CancellationToken ct
            = default)
    {
        for (int i = 0; ; i++)
        {
            ct.ThrowIfCancellationRequested();
            await Task.Delay(10, ct); // simulate
            I/O latency
            yield return i;
        }
    }
}
```

```

using Xunit;

public class TelemetryTests
{
    [Fact]
    public async Task
        Stream_throws_when_cancelled_mid_iteration()
    {
        using var cts = new CancellationTokensource();
        var seen = new List<int>();

        // OperationCanceledException (which
        // TaskCanceledException derives from)
        // must surface out of the await foreach, not
        // be swallowed. The producer
        // resumes after `yield return 2`, hits
        // `ThrowIfCancellationRequested` at
        // the top of the next iteration, and throws
        // the *base* type – so we
        // assert OperationCanceledException, not the
        // derived TaskCanceledException
        // (which only Task.Delay would throw, and we
        // never reach it post-cancel).
        Func<Task> act = async () =>
        {
            await foreach (var reading in
                Telemetry.ReadingsAsync(cts.Token))
            {
                seen.Add(reading);
                if (reading == 2) cts.Cancel(); //
                trip cancellation mid-stream
            }
        };

        await
        Assert.ThrowsAsync<OperationCanceledException>
            (act);
        Assert.Equal([0, 1, 2], seen);           //
        collection expression, C# 12

    }

    [Fact]
    public async Task
        Stream_honors_WithCancellation_on_the_consumer
        _side()
    {
        using var cts = new
            CancellationTokensource(TimeSpan.FromMilliseco
            nds(35));
    }
}

```



```

        await
        Assert.ThrowsAnyAsync<OperationCanceledException>(async () =>
        {
            // WithCancellation flows the token into
            // [EnumeratorCancellation]
            // even though the producer was called
            // with default.
            await foreach (var _ in
            Telemetry.ReadingsAsync().WithCancellation(cts
            .Token))
            {
            }
        });
    }
}

```

Two things matter here. First, assert on the *base* `OperationCanceledException` (or `ThrowsAnyAsync`) unless you specifically depend on `TaskCanceledException` — `Task.Delay` throws the latter, but a manual `ThrowIfCancellationRequested` throws the former, and over-specific assertions break when the implementation's cancellation point moves. Second, `WithCancellation` is the consumer-side mechanism that feeds the `[EnumeratorCancellation]` parameter; a test that only ever passes the token to the producer call misses the very common case where a caller has the enumerable but not the producer.

There is a subtler correctness property worth a dedicated test: *prompt* cancellation versus *eventual* cancellation. A stream that checks the token only at the top of each loop iteration cannot interrupt a long `await` already in flight — cancellation latency equals the remaining duration of the current operation. Passing the token *into* every awaited call (here `Task.Delay(10, ct)`) is what makes cancellation prompt; omit it and the token is only observed between iterations. When you test a real I/O stream, assert not just *that* it cancels but that it cancels *within a bounded number of items*, because “it eventually stops” is a much weaker guarantee than most callers assume. The asymmetry also bites resource cleanup: an `await foreach` compiles to a `try/finally` that calls `DisposeAsync` on the enumerator, so a generator holding a file handle or network socket must place its

cleanup in a `finally` inside the iterator — a `yield break` after an exception will not run code that sits after the `yield` statements. Test disposal explicitly by enumerating manually with `await using var e = source.GetAsyncEnumerator(ct)` and verifying the underlying resource was released.

Avoid real time in tests. `Task.Delay(10)` above keeps the example readable, but in a real suite use `Microsoft.Extensions.Time.Testing.FakeTimeProvider` (the BCL `TimeProvider` abstraction, GA since .NET 8) so you can advance a virtual clock deterministically and never pay wall-clock latency or court flakiness from a loaded CI box. A test that waits 35 real milliseconds to observe a timeout is a test that occasionally waits 60 on a contended runner and fails; replacing wall-clock delays with `FakeTimeProvider.Advance(TimeSpan)` removes the timing race entirely and makes the test both faster and deterministic. The same principle covers `PeriodicTimer`, retry/backoff loops, and circuit breakers — anything time-dependent should consume an injected `TimeProvider` so the test owns the clock.

Testing `Span<T>` and `ref struct` APIs

`ref struct` types — `Span<T>`, `ReadOnlySpan<T>`, `ref struct` you define yourself — cannot live on the heap, cannot be captured by a lambda or closure, and (until C# 13) could not appear in `async` methods or iterators. That last point is the crux for testing: **you cannot `await` across a line that holds a `Span` local**, and you cannot capture a `Span` in the lambda you pass to `Assert.Throws`. So Span tests are synchronous, and assertions on Span-producing code must materialize results before crossing an `await` or a delegate boundary.

C# 13 relaxed two relevant constraints: the `allows ref struct` anti-constraint lets generic code accept ref structs as type arguments, and ref structs may now implement interfaces. C# 14 promoted the array/string → `Span<T>` / `ReadOnlySpan<T>` conversions from library-defined operators to *first-class language conversions*: they were already implicit when passing an array to a

single span parameter, but C# 14 makes them participate in generic type inference, overload resolution, and extension-method receiver binding — so calling a `ReadOnlySpan<char>` extension directly on a `string[]` element or array now just works. None of that makes spans heap-friendly — the test ergonomics are unchanged.

```
public static class Csv
{
    // Parses "a,b,c" into trimmed fields, allocation-
    // free over the input.
    // params ReadOnlySpan<...> overloads (C# 13)
    // aren't needed here; this is
    // a plain span method that the test must drive
    // synchronously.
    public static int CountFields(ReadOnlySpan<char>
        line)
    {
        if (line.IsEmpty) return 0;
        int count = 1;
        foreach (var c in line)
            if (c == ',') count++;
        return count;
    }
}
```

```

public class CsvSpanTests
{
    [Theory]
    [InlineData("", 0)]
    [InlineData("solo", 1)]
    [InlineData("a,b,c", 3)]
    [InlineData("a,,c", 3)]
    public void CountFields_counts_correctly(string
        input, int expected)
    {
        // The span is created and consumed entirely
        within this synchronous
        // frame. No await, no lambda capture – both
        forbidden for ref structs.
        ReadOnlySpan<char> span = input;           //
        implicit conversion
        Assert.Equal(expected, Csv.CountFields(span));
    }

    [Fact]
    public void CountFields_works_over_a_stack_slice()
    {
        Span<char> buffer = stackalloc char[5];
        "a,b,c".CopyTo(buffer);
        Assert.Equal(3, Csv.CountFields(buffer)); //
        Span -> ReadOnlySpan
    }
}

```

If you must assert that a Span method throws, you cannot write `Assert.Throws<T>(() => Method(span))` — the span can't be captured. Materialize the throwing call into a local non-capturing helper, or convert inputs to arrays at the boundary and re-slice inside. This is the single most common compile error people hit when first testing Span APIs, and the fix is always “don't put the span in the closure.” A clean idiom is a `static` local function that takes the *array* and creates the span internally — the array is captured (legal), the span is not, and `Assert.Throws<ArgumentException>(() => ParseStrict(bytes))` compiles.

There is a deeper testing tradeoff for ref-struct-heavy code. Because spans can't cross `await`, can't be fields of a class, and can't be enumerated by LINQ, a suite built around mocking and async fixtures simply cannot reach into a span-based core. The

practical consequence is that you should keep ref-struct logic *pure and synchronous* — a function from `ReadOnlySpan<T>` to a value — and push the async/I/O orchestration to a thin outer layer. That separation is good design independent of testability, but Span makes it mandatory: the type system refuses to let you tangle a hot, allocation-free parser with the async plumbing around it. Test the pure core with synchronous `[Theory]` cases and the orchestration with async tests over a fake stream, and never try to make one test span both worlds. One more subtlety: when your `ref struct` is generic over a type that *might itself be a ref struct*, the C# 13 `allows ref struct` anti-constraint is what makes the generic instantiable with `Span<T>` as the type argument — test those instantiations explicitly, because a generic that compiles against `class` constraints can still fail to instantiate against a `ref struct` argument, and only an actual test that uses `Span<T>` as `T` proves the API surface is usable there.

Property-based testing and fuzzing for parsers and serializers

Example-based tests encode the cases you thought of. Parsers and serializers fail on the cases you didn't — and the strongest invariant for a serializer is the *round-trip*:

`deserialize(serialize(x)) == x` for all `x`. Property-based testing generates hundreds of random `x`, checks the invariant, and on failure *shrinks* the counterexample to a minimal reproduction. FsCheck (v3, the current generation, integrates with xUnit via `FsCheck.Xunit`) is the standard tool.

```

using FsCheck;
using FsCheck.Fluent;           // Prop, ArbMap, .Label()
    / fluent property builders
using FsCheck.Xunit;           // [Property]
using Xunit;                   // [Fact]
using System.Text.Json;

public record Point(int X, int Y, string Label);

public class RoundTripProperties
{
    // [Property] turns a method into a property-based
    // test. FsCheck supplies
    // random Point instances (record => structural
    // generator is derived).
    [Property(MaxTest = 500)]
    public Property Json_round_trips_any_point(Point
        original)
    {
        var json = JsonSerializer.Serialize(original);
        var restored =
            JsonSerializer.Deserialize<Point>(json);

        // Returning a Property (not bool) lets us
        // attach a label that prints
        // on shrink failure – invaluable for
        // diagnosing the minimal case.
        return (original == restored)
            .Label($"original={original} json={json}
                restored={restored}");
    }

    // A targeted property that drives FsCheck
    // explicitly (a plain [Fact] that
    // builds and runs the property itself), focusing
    // on string labels that
    // include control chars and quotes – inputs that
    // break naive serializers.
    // In FsCheck v3's C# (Fluent) API the default
    // arbitrary for a type comes
    // from ArbMap.Default.ArbFor<T>(); the v1/v2
    // `Arb.From<T>()` was renamed.
    [Fact]
    public void Json_handles_nasty_labels()
    {
        Prop.ForAll(
            ArbMap.Default.ArbFor<string>(), //
            strings incl. \0, \", emoji
            label =>
            {
                var p = new Point(0, 0, label ?? "");
            }
        );
    }
}

```

```

        var rt =
            JsonSerializer.Deserialize<Point>(
                JsonSerializer.Serialize(p));
        return rt == p;
    })
    .QuickCheckThrowOnFailure();           //
    throws -> xUnit sees failure
}

```

When the round-trip fails, FsCheck doesn't just hand you a 4,000-character random string — it shrinks to something like `Label = "\0"` or `X = -1`, the smallest input that still breaks. That shrinking is the entire value proposition; a hand-rolled `Random` loop gives you a failing seed and no minimization.

Round-trip is the headline property, but it is not the only one worth encoding. Useful serializer invariants include *idempotence* (`format(parse(format(x))) == format(x)`, which catches non-canonical output), *order independence* for set-like structures, *length/size monotonicity* for length-prefixed formats, and *negative properties* such as “malformed input never throws anything other than the one documented exception type.” Each of these is a one-line predicate over generated input and each catches a distinct class of bug that example tests routinely miss. Tune the generators, too: FsCheck's default `string` generator already includes nulls, surrogate pairs, and control characters, but for a numeric parser you will want to register an `Arbitrary` that biases toward boundary values (`int.MinValue`, `0`, `-1`, values straddling `Int32` / `Int64` limits) because uniform random integers almost never land on the off-by-one that breaks a hand-written parser. The shrinker is configurable as well; a custom shrinker that knows your domain produces minimal counterexamples in *your* terms (a minimal failing AST, not a minimal failing string).

For binary formats and untrusted input, go further and **fuzz**. `SharpFuzz` (libFuzzer bindings for .NET) drives your parser with coverage-guided mutation and is the right tool for “this must never throw an unexpected exception or read out of bounds.” A property test answers “is my logic correct?”; a fuzzer

answers “can hostile input crash me?” The distinction matters for *where* each runs: property tests are fast and deterministic enough to gate every PR (a fixed seed makes the run reproducible), whereas fuzzers explore for minutes-to-hours guided by coverage feedback, so they belong in a nightly job whose job is to *discover* new corpus entries. The healthy pipeline feeds discovered fuzz crashers back as permanent regression cases: when the fuzzer finds an input that throws

`IndexOutOfRangeException`, you minimize it, add it as an `[InlineData]` example, fix the bug, and the fast suite now guards that exact shape forever. Treat the fuzzer as a generator of *new test cases*, not as a gate, and you get the best of both — broad exploration off the critical path, tight verification on it.

BenchmarkDotNet fundamentals

`Stopwatch` in a loop is how almost everyone first measures performance, and it is almost always wrong. It conflates JIT warmup (the first call is interpreted/tier-0, then re-JITted at tier-1), background GC, OS scheduling jitter, and the cost of the `Stopwatch` reads themselves. You measure noise and warmup, then ship a “10× faster” claim built on a single unstabilized sample. **BenchmarkDotNet** exists to remove every one of those confounds: it runs in a separate optimized process, performs pilot runs to size the iteration count, warms up until timings stabilize, runs many measured iterations, and reports mean *with* a confidence interval and standard deviation.

Here is the canonical shape — two implementations, one marked baseline, with memory tracked:


```

using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System.Text;

[MemoryDiagnoser] // report
allocations + GC counts
public class StringJoinBenchmarks
{
    private readonly string[] _parts =
        Enumerable.Range(0, 64).Select(i => $"item-
{i}").ToArray();

    [Benchmark(Baseline = true)]
    public string Concat_PlusEquals()
    {
        var s = "";
        foreach (var p in _parts) s += p + ","; //
        O(n^2) allocations
        return s;
    }

    [Benchmark]
    public string StringBuilder_Build()
    {
        var sb = new StringBuilder();
        foreach (var p in _parts)
            sb.Append(p).Append(',');
        return sb.ToString();
    }

    [Benchmark]
    public string String_Join() => string.Join(',',
        _parts) + ",";
}

public static class Program
{
    public static void Main(string[] args) =>
        BenchmarkRunner.Run<StringJoinBenchmarks>();
}

```

A representative run (numbers illustrative — yours will differ by hardware):

Method	Mean	Ratio	Allocated
Concat_PlusEquals	1,412.0 ns	1.00	9,512 B
StringBuilder_Build	268.4 ns	0.19	1,560 B
String_Join	201.7 ns	0.14	880 B

The `Ratio` column is what you report: `string.Join` is ~7× faster and allocates ~10× less than naive concatenation. `[MemoryDiagnoser]` (built-in, cross-platform, not enabled by default) reports allocated bytes at ~99.5% accuracy and the number of GC collections per generation — frequently more decision-relevant than nanoseconds, because allocation drives GC pressure that doesn’t show up in a single benchmark’s mean. A method that allocates a kilobyte per call looks cheap in isolation — the gen-0 collection it triggers happens *amortized*, charged to some unlucky later call — but at scale that allocation rate determines how often the GC runs, how long pauses last, and whether you can use the server GC’s background mode effectively. The “Allocated” column makes that invisible cost visible per operation, which is exactly why allocation regressions are often a better CI gate than time regressions: bytes-allocated is deterministic and machine-independent, while nanoseconds are neither.

Note also *what* `[MemoryDiagnoser]` does and does not see. It measures managed heap allocations via GC bookkeeping; it does not see `stackalloc`, `ArrayPool<T>` rentals (which allocate once and reuse), or native memory. That is a feature — a benchmark showing “o B allocated” after you switch a hot path to `stackalloc` plus pooled buffers is precisely the proof you want — but it means “o B” is a statement about *managed heap* pressure, not total memory traffic. Pair it with `[ThreadingDiagnoser]` when contention is the suspect, since a method can be allocation-free and still slow because threads serialize on a lock.

Jobs and diagnosers

A *job* configures the runtime and run strategy. Use

`[SimpleJob]` to run across multiple runtimes in one pass, and the attribute-based diagnosers to attach instrumentation:

```
using BenchmarkDotNet.Diagnosers;
using BenchmarkDotNet.Jobs;

[MemoryDiagnoser]
[DisassemblyDiagnoser(maxDepth: 2, printSource: true)]
    // emit asm + C#
[SimpleJob(RuntimeMoniker.Net90, baseline: true)]
[SimpleJob(RuntimeMoniker.Net10_0)]
    // compare runtimes
public class CrossRuntimeBenchmarks
{
    // ... benchmarks run on both .NET 9 and .NET 10,
    // with .NET 9 as baseline
}
```

`[ThreadingDiagnoser]` reports lock contention and completed work items; `[EventPipeProfiler]` captures a trace you can open in Speedscope or PerfView. Reach for the diagnoser that answers your specific question — attaching all of them slows the run and clutters the report.

Defeating dead-code elimination

The JIT is allowed to delete any computation whose result is never observed. If your benchmark computes a value and discards it, tier-1 codegen may erase the whole method body, and you'll “measure” a few nanoseconds of nothing. The fix is to **consume the result** — return it from the method (BenchmarkDotNet feeds returned values to a consumer), or use `Consumer.Consume` for loops that produce many values:

```

using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Engines;

public class ConsumptionBenchmarks
{
    private readonly int[] _data = Enumerable.Range(0,
        10_000).ToArray();
    private readonly Consumer _consumer = new();

    // WRONG: result discarded. JIT may elide the
    //         entire Sum.
    [Benchmark]
    public void Discarded()
    {
        long _ = _data.Sum(x => (long)x); // dead
        store -> may vanish
    }

    // RIGHT: returning the value forces the
    //         computation to be observed.
    [Benchmark]
    public long Returned() => _data.Sum(x => (long)x);

    // RIGHT for loops producing many values: feed
    //         each to a Consumer,
    //         which is a sink the JIT cannot prove is side-
    //         effect-free.
    [Benchmark]
    public void LoopConsumed()
    {
        foreach (var x in _data)
            _consumer.Consume(x * x);
    }
}

```

If `Discarded` reports a time suspiciously close to your empty-method overhead, or far below `Returned`, dead-code elimination is the prime suspect. Always sanity-check that the two implementations you’re comparing report plausible, *similar-order* costs unless you expect a dramatic difference.

`[DisassemblyDiagnoser]` : proving inlining and vectorization

When the claim is “this hot path is inlined and vectorized,” don’t assert it from the source — read the codegen.

`[DisassemblyDiagnoser]` emits the actual machine code BenchmarkDotNet's optimized process produced.

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Diagnosers;
using System.Numerics;
using System.Runtime.CompilerServices;

[DisassemblyDiagnoser(maxDepth: 3, printSource: true)]
public class SumBenchmarks
{
    private int[] _data = null!;

    [GlobalSetup]
    public void Setup() => _data = Enumerable.Range(0,
        1024).ToArray();

    // Scalar loop. We expect the JIT to inline the
    // helper and possibly
    // auto-vectorize the accumulation.
    [Benchmark(Baseline = true)]
    public int ScalarSum()
    {
        int total = 0;
        foreach (var x in _data) total += x;
        return total;
    }

    // Explicit SIMD via Vector<T>. Disassembly should
    // show packed adds
    // (e.g. vpaddd / paddd on x64-AVX2) over the
    // data.
    [Benchmark]
    public int VectorSum()
    {
        var acc = Vector<int>.Zero;
        var span = _data.AsSpan();
        int i = 0;
        int width = Vector<int>.Count;
        for (; i <= span.Length - width; i += width)
            acc += new Vector<int>(span.Slice(i,
                width));
        int total = Vector.Dot(acc, Vector<int>.One);
        // horizontal reduce
        for (; i < span.Length; i++) total += span[i];
        // tail
        return total;
    }
}
```

In the disassembly output you're looking for two things in `VectorSum`: packed integer-add instructions (`vpaddq` / `paddd`) confirming the SIMD path was taken, and the absence of a `call` to `Vector<int>::op_Addition` — if the operator was inlined, you'll see the arithmetic inline rather than a function call. For `ScalarSum`, modern JITs frequently auto-vectorize a simple accumulation loop too, so don't be surprised if both show packed adds; the diagnoser tells you the *truth* about codegen, which often defies intuition. If you see a `call CORINFO_HELP_*` inside what you believed was a tight inlined loop, your inlining assumption was wrong — possibly the method exceeded the JIT's inline budget, in which case `[MethodImpl(MethodImplOptions.AggressiveInlining)]` is the lever, used sparingly.

The disassembly is also where you confirm or refute *defensive copies*, *bounds-check elimination*, and *covariant array checks* — the silent taxes that intuition can't see. A `readonly struct` passed by `in` should not be copied at the call site; if the JIT can't prove a member access is side-effect-free it will defensively copy the struct, and the only way to catch it is to see the copy in the asm. Span indexing inside a `for (i = 0; i < span.Length; i++)` loop usually has its bounds check *removed* because the JIT proves `i` is in range from the loop condition; rewrite the loop so the JIT loses that proof (e.g. iterate to a cached length that it can't correlate) and the bounds check reappears as a `cmp` / `jae` pair per iteration — visible only in disassembly. This is the value of `[DisassemblyDiagnoser]` over a benchmark mean: the mean tells you *that* something is slower; the asm tells you *why*. Reserve it for the handful of genuinely hot methods where the difference between a removed bounds check and a retained one is the difference you're shipping — running it on every benchmark is slow and the output is noise for code that isn't on the critical path. And remember the codegen you read is for *one* CPU's ISA; an AVX-512 machine and an Arm64 machine produce entirely different instructions for the same `Vector<T>` source, so a disassembly-backed claim is implicitly scoped to the architecture you ran on.

Reading output honestly: baselines, noise, and statistics

A benchmark mean without dispersion is a rumor.

BenchmarkDotNet reports `StdDev` and, on request

(`[StatisticalTestColumn]` or the `--statisticalTest` option), a significance test against the baseline. Interpretation discipline:

- **Look at the confidence interval, not the point estimate.** If two methods' intervals overlap heavily, "A is faster than B" is unsupported regardless of which mean is lower. A 2% difference with 5% standard deviation is noise.
- **Warnings are not decoration.** BenchmarkDotNet prints `MultimodalDistribution`, `ZeroMeasurement`, or `Outliers detected` when the data violates its assumptions. A multimodal distribution usually means a branch is data-dependent (e.g., a cache hit/miss split) — your single "mean" is hiding two populations.
- **Pin the environment.** Disable Turbo Boost / set the CPU governor to `performance`, close other processes, and never trust a benchmark run on a CI runner sharing a noisy host for absolute numbers — use it only for *relative* ratios against a same-run baseline.
- **`[Benchmark(Baseline = true)]` is the anchor.** Ratios computed within one run cancel out machine-specific constant factors, which is exactly why you compare against an in-run baseline rather than against numbers from last week's laptop.

Regression gates in CI

A benchmark you run once and forget is a vanity metric. To *defend* performance you need an automated gate: run the suite in CI, compare against a stored baseline, and fail the build if a key benchmark regresses beyond a threshold. The mechanism is to export results as JSON and diff them.

```
using BenchmarkDotNet.Configs;
using BenchmarkDotNet.Exporters.Json;
using BenchmarkDotNet.Running;

public static class CiGate
{
    public static void Main(string[] args)
    {
        var config = DefaultConfig.Instance
            .AddExporter(JsonExporter.FullCompressed);
        // writes
        BenchmarkDotNet.Artifacts/results/*.json
        BenchmarkRunner.Run<StringJoinBenchmarks>
            (config);
    }
}
```



```

// A minimal gate: load the just-produced report and
// the committed baseline,
// fail if any benchmark's mean regressed by more than
// the allowed fraction.
using System.Text.Json;

static class RegressionGate
{
    record Stat(double Mean);

    static int Main()
    {
        const double AllowedRegression = 0.10;
        // 10% tolerance band

        var current = Load("artifacts/current.json");
        var baseline =
            Load("artifacts/baseline.json");

        int failures = 0;
        foreach (var (name, cur) in current)
        {
            if (!baseline.TryGetValue(name, out var
            bas)) continue;
            double delta = (cur.Mean - bas.Mean) /
            bas.Mean;
            if (delta > AllowedRegression)
            {
                Console.Error.WriteLine(
                    $"REGRESSION {name}:
                    {bas.Mean:F1}ns -> {cur.Mean:F1}ns (+
                    {delta:P1})");
                failures++;
            }
        }
        return failures == 0 ? 0 : 1;
        // nonzero -> CI fails
    }

    static Dictionary<string, Stat> Load(string path)
    {
        using var doc =
            JsonDocument.Parse(File.ReadAllText(path));
        var map = new Dictionary<string, Stat>();
        foreach (var b in
            doc.RootElement.GetProperty("Benchmarks").EnumerateArray())
        {
            var name =
                b.GetProperty("FullName").GetString(!);

```

```

        var mean =
            b.GetProperty("Statistics").GetProperty("Mean")
              .GetDouble();
        map[name] = new Stat(mean);
    }
    return map;
}

```

Two hard-won cautions. First, set the tolerance generously (10–20%) because CI hardware is noisy; a tight gate produces false alarms that train the team to ignore it. The mature alternative is to compare *ratios* within a single run (a fixed reference benchmark vs. the method under test) so machine variance cancels — far more stable than absolute nanoseconds across machines. Second, for serious projects, prefer the purpose-built tooling: `crank` / `BenchmarkDotNet`’s integration with statistical regression detection, or the `ResultsComparer` tool the dotnet team ships, which runs a proper Mann–Whitney U test instead of a naive mean diff.

Profiling and correlating with benchmarks

Benchmarks tell you *which* implementation is faster on a micro-scopic, isolated workload. Profilers tell you *where* a whole application spends its time and allocates — and the two often disagree, because a method that’s 7× faster in isolation is irrelevant if it runs 0.01% of the time. Use benchmarks to optimize a hot path you’ve *already identified by profiling*, never the reverse. Premature microbenchmarking is how teams shave nanoseconds off code that doesn’t matter.

The dotnet diagnostic CLI tools are the in-the-box profilers and are cross-platform:

```
# Install once.
dotnet tool install --global dotnet-trace
dotnet tool install --global dotnet-counters

# Live counters: GC heap size, alloc rate, gen-0/1/2
# counts, thread pool
# queue length, exceptions/sec, lock contention -
# refreshed in place.
dotnet-counters monitor --process-id 12345 \
    --counters
    System.Runtime,Microsoft.AspNetCore.Hosting

# Capture a CPU sampling trace for 20 seconds, then
# open the .nettrace
# in PerfView / Speedscope / the VS profiler.
dotnet-trace collect --process-id 12345 --duration
    00:00:20 \
    --profile cpu-sampling --output app.nettrace
```

The correlation workflow: `dotnet-counters` shows an abnormal gen-0 collection rate under load → `dotnet-trace` with the GC profile reveals which call tree allocates → you reproduce that allocation in a focused BenchmarkDotNet harness with `[MemoryDiagnoser]` → you fix it (pool buffers, switch to `Span`, use `ArrayPool<T>`) and the benchmark’s “Allocated” column proves the fix → the regression gate locks it in. Each tool answers a different question; using a benchmark to find a hotspot, or a profiler to compare two implementations precisely, is using the wrong instrument.

Testing source generators and analyzers

Source generators and Roslyn analyzers are compiler plugins, and you test them by *running the compiler* over input code and asserting on the output — generated trees, emitted diagnostics, and that the result still compiles. The `Microsoft.CodeAnalysis.CSharp.SourceGenerators.Testing` and `...Analyzer.Testing` packages provide a verifier harness so you don’t hand-roll a `CSharpCompilation`.

```

using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Testing;
using Microsoft.CodeAnalysis.Testing;
using Xunit;

public class ToStringGeneratorTests
{
    [Fact]
    public async Task
        Generates_partial_with_member_dump()
    {
        const string input = """
            using Generators;
            [GenerateToString]
            public partial class Account
            {
                public int Id { get; set; }
                public string Name { get; set; } = "";
            }
            """;

        const string expectedHint = "Account.g.cs";
        const string expectedSource = """
            // <auto-generated>
            partial class Account
            {
                public override string ToString()
                    => $"Account {{ Id = {Id}, Name =
{Name} }}";
            }
            """;

        var test = new
            CSharpSourceGeneratorTest<ToStringGenerator,
            DefaultVerifier>
        {
            TestState =
            {
                Sources = { input },
                GeneratedSources =
                {
                    (typeof(ToStringGenerator),
                    expectedHint,
                    SourceText.From(expectedSource,
                    System.Text.Encoding.UTF8)),
                },
            },
        };
    }
}

```

```

        await test.RunAsync(); // fails if generated
                               text differs byte-for-byte
    }
}

```

For analyzers, the verifier asserts on diagnostics at specific source locations using the `{|RULE_ID:span|}` markup syntax:

```

using VerifyCS =
    Microsoft.CodeAnalysis.CSharp.Testing.CSharpAn
alyzerVerifier<
    MyAnalyzers.UseSpanAnalyzer,
    Microsoft.CodeAnalysis.Testing.DefaultVerifier
>;

public class UseSpanAnalyzerTests
{
    [Fact]
    public async Task Flags_substring_in_hot_loop()
    {
        // The {|MA0001:...|} markup asserts that
        // diagnostic MA0001 is reported
        // on exactly that span – location AND id are
        // verified.
        const string source = """
            class C
            {
                string Mid(string s) =>
                {|MA0001:s.Substring(1, 2)|};
            }
            """;
        await VerifyCS.VerifyAnalyzerAsync(source);
    }
}

```

Generator tests are exacting — they compare generated text byte-for-byte, so a stray trailing newline fails the test. That strictness is the point: it pins the generator’s output as a contract. Two pragmatic refinements. First, snapshot-test with `Verify` (the `Verify.SourceGenerators` package) when output is large; it diffs against an approved `.verified.cs` file and gives a clean review of intentional changes rather than a wall of string-mismatch. Second, always include a test that the *generated code compiles and behaves*, not merely that its text matches — assert `GeneratorDriverRunResult.Diagnostics` is empty

and, ideally, load the compiled assembly and exercise the generated member. Text-equality alone can pass while the generated code references a type you forgot to emit a `using` for.

A worked example: tying it together

Suppose you replace a LINQ-based parser hot path with a `Span`-based one. The full validation loop — correctness by property test, performance by benchmark, codegen by disassembly, durability by gate — looks like this. First the two implementations and a property test proving they agree:

```
public static class IntListParser
{
    // Old: allocates a string[] (Split), plus a Where
    //       iterator and a delegate.
    // long accumulation so it matches the span
    //       version's overflow behavior
    // exactly – Enumerable.Sum(int) is *checked* and
    //       throws on overflow, the
    // span loop's `+=` is *unchecked*; summing into
    //       long avoids the divergence.
    public static long SumLinq(string csv)
        => csv.Split(',').Where(s => s.Length >
            0).Sum(s => (long)int.Parse(s));

    // New: zero intermediate allocations, span
    //       slicing only.
    public static long SumSpan(ReadOnlySpan<char> csv)
    {
        long total = 0;
        foreach (var range in csv.Split(','))
            // ReadOnlySpan.Split, .NET 9+
            {
                var field = csv[range].Trim();
                if (!field.IsEmpty) total +=
                    int.Parse(field);
            }
        return total;
    }
}
```

```

using FsCheck;
using FsCheck.Fluent;    // .Label() extension on
                        bool/Property
using FsCheck.Xunit;

public class ParserEquivalence
{
    // Property: the two implementations must agree on
    // every list of ints.
    // FsCheck generates int[]; we format to CSV and
    // check both parsers match.
    [Property(MaxTest = 1000)]
    public Property Span_and_Linq_agree(int[] values)
    {
        var csv = string.Join(',', values);
        long linq = IntListParser.SumLinq(csv);
        // The implicit string -> ReadOnlySpan<char>
        // conversion is long-standing
        // (since Span shipped, .NET Core 2.1 / C#
        // 7.2) – not a C# 14 feature.
        long span = IntListParser.SumSpan(csv);
        return (linq == span).Label($"csv="{csv}"\n"
            linq={linq} span={span}");
    }
}

```

```

using BenchmarkDotNet.Attributes;

[MemoryDiagnoser]
public class ParserBenchmarks
{
    private string _csv = null!;

    [GlobalSetup]
    public void Setup() =>
        _csv = string.Join(',', Enumerable.Range(0,
            256));

    [Benchmark(Baseline = true)]
    public long Linq() => IntListParser.SumLinq(_csv);
    // result returned -> consumed

    [Benchmark]
    public long Span() => IntListParser.SumSpan(_csv);
}

```

The property test (1,000 random integer arrays, shrinking on failure) proves behavioral equivalence before you trust the faster

path — if the Span version mishandles, say, a leading minus sign or `int.MinValue`, FsCheck finds and minimizes it. The benchmark, with both results *returned* (so the JIT can't elide them) and `[MemoryDiagnoser]` attached, proves the `Span` version is faster and — the headline — allocates zero bytes versus the LINQ version's `string[]` plus enumerator and delegate allocations. Drop the exported JSON into the regression gate from the previous section, and a future “harmless refactor” that reintroduces `Split` into a `string[]` fails CI on the Allocated-bytes delta. That is the whole loop: a claim (“faster and allocation-free”), proven by measurement, defended automatically — correctness and performance both backed by evidence rather than hope.

Part VIII — Reference

30 · Capstone Reference: What's New in C# 13 and C# 14

This chapter consolidates the language surface added across the two releases that bracket .NET 9 and .NET 10. It is built to be read either linearly — to fold the new features into a coherent mental model — or used as a lookup table when you are mid-migration and need to know exactly which version, SDK, and target framework a given construct requires. Every feature below is attributed to the C# version that shipped it; nothing here is speculative.

Version map at a glance

The single most common source of confusion when adopting modern C# is conflating *compiler* support with *runtime* support. The C# compiler is delivered by the SDK and gated by `<LangVersion>`; some features compile against any TFM, but a handful lower to BCL types or runtime behaviors that only exist on a recent target framework.

C#	Ships with	First Visual Studio	Headline additions
12	.NET 8 (Nov 2023)	VS 2022 17.8	Primary constructors (all types), collection expressions, <code>ref readonly</code> params, default lambda params, <code>using</code> aliases for any type, inline arrays
13	.NET 9 (Nov 2024)	VS 2022 17.12	<code>params</code> collections, <code>System.Threading.Lock</code> , <code>\e</code> escape, <code>ref / unsafe</code> in iterators & async, <code>allows ref struct</code> , ref struct interfaces, implicit-index in initializers, <code>[OverloadResolutionPriority]</code> , partial properties/indexers
14	.NET 10 (Nov 2025)	VS 2026	<code>field</code> keyword, extension members (extension blocks), null-conditional assignment, <code>nameof</code> of unbound generics, first-class <code>Span<T></code> conversions, partial constructors & events, user-defined compound assignment, modifiers on simple lambda params

`<LangVersion>` accepts `13`, `14`, `latest`, `latestMajor`, `preview`, and `default`. The default for a project is derived from its `<TargetFramework>`: `net9.0` defaults to C# 13, `net10.0` to C# 14. You can raise `<LangVersion>` ahead of your TFM — for example C# 14 syntax on `net8.0` — but only for features whose lowering does not depend on a newer BCL. The two practical hard gates are covered under the migration checklist below.

C# 13: `params` collections

Before C# 13, `params` was restricted to arrays. C# 13 generalizes it to `Span<T>`, `ReadOnlySpan<T>`, `IEnumerable<T>`, and any type the collection-expressions machinery can build —

`List<T>`, `IReadOnlyList<T>`, interfaces, and types with a `[CollectionBuilder]` attribute.

The performance story is the reason to care. A `params` `ReadOnlySpan<T>` overload lets the compiler stack-allocate the argument buffer at the call site, eliminating the heap array that `params T[]` forced on every variadic call.

```
using System;
using System.Collections.Generic;

static int Sum(params ReadOnlySpan<int> values) // C#
    13: params + ReadOnlySpan<T>
{
    int total = 0;
    foreach (int v in values) total += v;
    return total;
}

static void Log(params IEnumerable<string> lines) //
    C# 13: params + IEnumerable<T>
{
    foreach (var line in lines)
        Console.WriteLine(line);
}

// Call site: no int[] is allocated; the buffer lives
// on the stack.
int s = Sum(1, 2, 3, 4);           // s == 10, zero
// heap allocations
Log("a", "b", "c");
```

When you provide multiple `params` overloads (say, both `ReadOnlySpan<T>` and `IEnumerable<T>`), overload resolution prefers the span form for direct variadic calls because it is the more specific, allocation-free path. This is exactly why .NET 9 retrofitted `params ReadOnlySpan<T>` onto 60-plus existing variadic BCL methods (`string.Join`, `string.Format`, `StringBuilder.Append*`, and many more). Note the matching behavior change: when both a `params T[]` and a `params` `ReadOnlySpan<T>` overload exist, C# 13 overload resolution now prefers the span overload — a documented breaking change that, for instance, surfaces inside `Expression` trees, which cannot host the `ref struct` span. Use `IEnumerable<T>` only when

callers genuinely need to forward an existing lazy sequence; otherwise the span overload is strictly better.

C# 13: the `Lock` type and the `lock` pattern

C# 13 introduced `System.Threading.Lock`, a dedicated mutual-exclusion type that replaces the historical pattern of `lock` ing on a bare `object`. The compiler special-cases it: `lock (myLock) { ... }` on a `Lock`-typed variable lowers to `myLock.EnterScope()` returning a `ref struct Scope` that is disposed at block exit, rather than to `Monitor.Enter / Monitor.Exit`.

```
using System.Threading;

private readonly Lock _gate = new(); // C# 13 + .NET 9: System.Threading.Lock

public void Mutate()
{
    lock (_gate) // lowers to using var scope = _gate.EnterScope();
    {
        // critical section
    }
}
```

This is a .NET 9 -and-later feature: `System.Threading.Lock` is a runtime type, so it requires targeting `net9.0` +, not merely raising `LangVersion`. The payoff is a faster, allocation-lean lock with a clearer contract, and a type system that prevents the classic bug of accidentally `lock` ing on a value that escapes to other code. One gotcha: if a variable is *typed* as `object` but holds a `Lock` instance, the compiler emits a warning and falls back to `Monitor`-based locking — keep the static type as `Lock` to get the fast path.

C# 13: the `\e` escape sequence

A small but welcome addition: `\e` is the ESC character (U+001B), previously spelled `\u001b` or `\x1b`. The latter, `\x1b`, is famously dangerous because `\x` consumes *one to four*

hex digits greedily — in a literal like `"\x1bExit"` it keeps eating into `1bE` (all valid hex digits) and yields the wrong code point. `\u001b` is unambiguous but verbose; `\e` is both short and unambiguous.

```
// C# 13: \e == ESC (U+001B); safer than \x1b which  
greedily eats hex digits  
const string Red = "\e[31m";  
const string Reset = "\e[0m";  
Console.WriteLine($"{Red}error{Reset}");
```

C# 13: `ref` locals and `unsafe` in iterators and `async`

Pre-C# 13, you could not declare a `ref` local or `ref struct` local, nor write an `unsafe` block, inside an `async` method or an iterator (`yield` -bearing method). The state machine rewrite could not safely hoist a `ref` across an `await` or `yield` suspension point. C# 13 relaxes this: `ref` locals and `unsafe` blocks are now allowed in those methods *as long as they do not live across a suspension point*. The compiler still rejects a `ref` local that would need to survive an `await` .

```

async Task<int> ProcessAsync(int[] data)
{
    int local = 0;
    {
        ref int slot = ref data[0]; // C# 13: ref
                                   // local inside async, used and finished
        slot = 42;                  // ...before any
                                   // await - legal
        local = slot;
    }
    await Task.Yield();             // ref local is
                                   // out of scope here
    return local;
}

IEnumerable<int> Doubled(int[] xs)
{
    foreach (int x in xs)
    {
        ref readonly int r = ref xs[0]; // C# 13: ref
                                         // local inside an iterator
        yield return x * 2 + r * 0;
    }
}

```

C# 13: allows ref struct and ref struct interfaces

Two related generics features landed together. First, `ref struct` types (like `Span<T>`) can now implement interfaces. Second, a new *anti-constraint*, where `T : allows ref struct`, lets a generic type parameter be substituted with a `ref struct`. The compiler then enforces ref-safety rules on `T` inside that generic — `T` cannot be boxed, stored on the heap, or used where it might escape.

```

using System;

interface IParser<T> { T Parse(ReadOnlySpan<char>
                             input); } // ref struct can implement this

// C# 13: allows ref struct anti-constraint
static T Run<TParser, T>(TParser p, ReadOnlySpan<char>
                        text)
    where TParser : IParser<T>, allows ref struct
    => p.Parse(text);

```

`allows ref struct` is what makes generic algorithms over `Span<T>` possible — for instance, the .NET 9 generic math and `IUtf8SpanParsable<T>` plumbing relies on it. Use it when you want to write `T`-generic code that should also accept stack-only types; do *not* reach for it casually, because every use of `T` inside the method is then constrained to ref-safe operations, which can be surprisingly limiting.

C# 13: implicit index access,

`[OverloadResolutionPriority]` , partial properties

Three quality-of-life additions:

Implicit index access in object initializers — the `^` from-end index operator is now usable inside an object/collection initializer, where previously only explicit indices were allowed.

`[OverloadResolutionPriority]` — lets a library author bias overload resolution toward a preferred overload (typically a new span-based one) without breaking source compatibility. Higher priority wins among otherwise-applicable candidates. This is a library-author tool; application code rarely declares it but is affected by it.

Partial properties and indexers — `partial` was extended from methods to properties and indexers, completing the source-generator story so a generator can supply the implementing accessor for a property declared in your hand-written partial class.


```

using System.Runtime.CompilerServices;

class Buffer
{
    public int[] Data { get; } = new int[5];

    // C# 13: implicit ^ index inside a nested object
    // initializer.
    // 'Data = { [^1] = 99 }' assigns into the
    // existing array via its indexer
    // (it does not reassign Data), so the get-only
    // property is fine.
    public static Buffer Create() => new Buffer { Data
        = { [^1] = 99 } }; // sets Data[4] = 99
}

partial class Vm
{
    public partial string Name { get; set; } // C# 13:
        partial property (declaring)
}

partial class Vm
{
    private string _name = ""; //
        explicit backing field (field keyword is C#
        14)
    public partial string Name //
        implementing part (e.g. from a generator)
    {
        get => _name;
        set => _name = value?.Trim() ?? "";
    }
}

static class Api
{
    [OverloadResolutionPriority(1)] //
        C# 13: preferred overload
    public static void Write(System.ReadOnlySpan<char>
        s) { }
    public static void Write(string s) { } //
        lower priority
}

```

Every C# 13 feature in one file

```
// ===== C# 13 (.NET 9) feature roll-up =====
using System;
using System.Collections.Generic;
using System.Threading;
using System.Runtime.CompilerServices;

class CSharp13Tour
{
    // params collections
    static int Sum(params ReadOnlySpan<int> xs) { int
        t = 0; foreach (var x in xs) t += x; return t;
    }

    // System.Threading.Lock + lock pattern
    static readonly Lock Gate = new();

    // \e escape sequence
    const string Esc = "\e[0m";

    // allows ref struct anti-constraint + ref struct
    // interface
    interface IReader { int Read(ReadOnlySpan<byte>
        b); }
    static int Use<TR>(TR r, ReadOnlySpan<byte> b)
        where TR : IReader, allows ref struct =>
        r.Read(b);

    // [OverloadResolutionPriority]
    [OverloadResolutionPriority(1)] static void
        Emit(ReadOnlySpan<char> s) { }
    static void Emit(string s) { }

    // ref local inside an iterator (scoped so it does
    // not live across the yield boundary)
    static IEnumerable<int> Iter(int[] a)
    {
        { ref int r = ref a[0]; r++; } // ref local
        // used and finished before yield
        yield return a[0];
    }

    // partial property
    public partial string Tag { get; set; }

    static void Main()
    {
        _ = Sum(1, 2, 3); //
        // stack buffer, no array alloc
    }
}
```

```

        lock (Gate) { Console.Write(Esc); } //
        EnterScope()-based lock
        var dict = new Dictionary<int, int> { [0] = 1
        }; // explicit indexer in an
        initializer
        var buf = new Holder { Slots = { [^1] = 99 }
        }; // C# 13: implicit ^ index in an
        initializer
        _ = (dict, buf);
    }
}

partial class CSharp13Tour
{
    private string _tag = ""; // explicit backing
    field – the field keyword is C# 14
    public partial string Tag { get => _tag; set =>
        _tag = value ?? ""; }
}

class Holder { public int[] Slots { get; } = new
    int[3]; } // target for the ^-index
    initializer

```

C# 14: the `field` keyword

The single most impactful day-to-day change in C# 14 is the `field` contextual keyword. Inside a property accessor, `field` refers to the compiler-synthesized backing field, so you can add logic to one accessor without hand-writing the field or the other accessor.

```

// C# 14: field-backed property – no explicit backing
// field
public string Message
{
    get;
    // auto-implemented getter
    set => field = value ?? throw new
        ArgumentNullException(nameof(value));
}

```

The getter here is still auto-implemented, and `field` in the setter binds to the *same* synthesized field the getter reads. You can give a body to one accessor or both; any accessor you leave as `get;` / `set;` shares the field. This collapses the most common reason teams wrote full properties — validation, normalization,

lazy defaults, change notification — down to a single line per accessor.

A worked normalization-and-notification example:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

class Person : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler?
        PropertyChanged;

    public string FirstName
    {
        get => field ?? ""; //
        // field may be null until first set
        set
        {
            string v = value.Trim();
            if (field == v) return; //
            // skip no-op notifications
            field = v; //
            // C# 14: assign synthesized backing field
            PropertyChanged?.Invoke(this,
                new(nameof(FirstName)));
        }
    }

    // Lazy default with no separate field:
    public int Cached
    {
        get => field is 0 ? field = Compute() : field;
        private set => field = value;
    }
    static int Compute() => 42;
}
```

field migration: side-by-side

Converting a class full of manual backing fields is mechanical. The diff below shows the before/after for a typical view-model property cluster:

```
// ----- BEFORE (C# 11/12): manual backing fields -----
class Settings
{
    private int _volume;
    public int Volume
    {
        get => _volume;
        set => _volume = Math.Clamp(value, 0, 100);
    }

    private string _theme = "light";
    public string Theme
    {
        get => _theme;
        set => _theme = value ?? "light";
    }

    private bool _dirty;
    public bool Dirty { get => _dirty; set => _dirty = value; }
}

// ----- AFTER (C# 14): field keyword -----
class Settings
{
    public int Volume
    {
        get;
        set => field = Math.Clamp(value, 0, 100);
    }

    public string Theme
    {
        get => field ?? "light";
        set => field = value ?? "light";
    }

    public bool Dirty { get; set; } // trivial
                                property collapses entirely
}

```

Note that `Theme`'s default (`= "light"`) moved into the getter's `?? "light"`. A `field`-backed property *can* still carry an initializer — you would write `public string Theme { get => field; set => field = value ?? "light"; } = "light";`, and the `= "light"` initializes the synthesized backing field —

but here folding the default into the getter's `??` keeps the property to two lines and avoids a separate initializer. Either form is correct; the semantics are identical and the field count drops from three to zero.

`field` compatibility note: a real breaking change

`field` is *contextual*, which means an existing identifier literally named `field` changes meaning the moment a property accessor references it. This is the only `field`-related break you must audit before flipping `LangVersion` to 14.

```
class Form
{
    private string field = "legacy-identifier"; // a
        real variable named 'field'

    // Pre-C#14 this read the instance variable above.
    // Under C#14 LangVersion, bare 'field' inside an
        accessor now means the
    // synthesized backing field of Label – NOT this
        variable.
    public string Label
    {
        get => field;           // AMBIGUITY: now binds
            to the synthesized backing field
        set => this.field = value; // use this.field
            (or @field) to reach the variable
    }
}
```

Inside the `Label` accessors, a bare `field` now resolves to the property's synthesized backing field, silently changing behavior. The fixes are `this.field` or the verbatim `@field` to mean the variable, or — far better — rename the variable. The Roslyn analyzer flags this, but only if you run analysis; a blind `LangVersion` bump can ship the regression. Grep for the identifier `field` across the codebase before upgrading.

C# 14: extension members

C# 14 generalizes extension methods into *extension members* via the `extension` block. Inside a top-level non-generic static

class, an `extension(ReceiverType receiver)` block can declare extension **methods, properties, and operators** — both instance-style (operating on `receiver`) and static-style (operating on the type itself).

```
using System;
using System.Collections.Generic;
using System.Linq;

public static class SequenceExtensions
{
    // Instance extension block: members appear on an
    // IEnumerable<T> instance
    extension<T>(IEnumerable<T> source)
    {
        public bool IsEmpty => !source.Any();
        // C# 14: extension property
        public int CountFast => source is
            ICollection<T> c ? c.Count : source.Count();
    }

    // Static extension block: members appear as
    // static members of the type
    extension<T>(IEnumerable<T>)
    {
        public static IEnumerable<T> Identity =>
            Enumerable.Empty<T>(); // static ext. property
        // static extension (user-defined) operator
        public static IEnumerable<T> operator +
            (IEnumerable<T> a, IEnumerable<T> b) =>
            a.Concat(b);
    }
}

// Usage:
List<int> nums = [1, 2, 3];
bool empty = nums.IsEmpty; //
    instance-style extension property -> false
var combined = nums + new[] { 4, 5 }; // static
    extension operator -> 1,2,3,4,5
var id = IEnumerable<int>.Identity; // static
    extension property
```

The receiver in a static-only block needs no name (`extension<T>(IEnumerable<T>)`); if you supply one it goes unused. Under the hood, each extension member lowers to a plain static method that takes the receiver as its first parameter — a property getter becomes a static

`get_IsEmpty(IEnumerable<T> source)` -shaped method — so the cost matches a classic extension method: zero allocation, statically dispatched, and inline-friendly. The big wins are extension *properties* (impossible before C# 14) and extension *operators*, which let you add, for example, a `+` operator to a type you do not own.

A caveat worth internalizing: extension members do not participate in interface satisfaction or virtual dispatch. They are syntactic sugar resolved at compile time against the static type of the receiver, exactly like classic extension methods. Do not reach for them expecting polymorphism.

C# 14: null-conditional assignment

`?.` and `?[]` may now appear on the **left** of `=` and compound assignments. If the receiver is null, the whole assignment is skipped — *and the right-hand side is not evaluated*.

```
class Order { public decimal Total { get; set; } }
class Customer { public Order? Pending { get; set; } }

Customer? c = GetCustomer();

c?.Pending = new Order();           // C# 14:
    assignment skipped entirely if c is null
c?.Pending?.Total += Surcharge();   // compound
    assignment also supported

static Customer? GetCustomer() => null;
static decimal Surcharge() => throw new
    InvalidOperationException("must not run");
```

The pitfall, called out explicitly because it bites people: this is **not** pure syntactic sugar for “assign if not null.” The right-hand side short-circuits. In `c?.Pending?.Total += Surcharge()`, when `c` (or `c.Pending`) is null, `Surcharge()` is *never called*. If you were relying on a side effect in the RHS firing unconditionally, this changes behavior. `++` and `--` are deliberately not allowed in this position. Treat the RHS as living inside the same conditional as the receiver — because it does.

C# 14: `nameof` with unbound generic types

`nameof` now accepts an *unbound* generic type, returning the bare type name. Previously you had to close the type — `nameof(List<int>)` — even though the type argument was irrelevant to the result.

```
using System.Collections.Generic;

static class Diagnostics
{
    public static string Describe() =>
        // C# 14: nameof on unbound generics in a
        // diagnostic message
        $"Collection types: {nameof(List<>)},
        {nameof(Dictionary<,>)},
        {nameof(IReadOnlyList<>)}"
        // => "Collection types: List, Dictionary,
        //      IReadOnlyList"
}
```

This is purely a compile-time string operation (the result is `"List"`, `"Dictionary"`, `"IReadOnlyList"`), so it carries zero runtime cost. It is most valuable in logging, analyzer diagnostics, and source generators where you want the type's simple name without inventing arbitrary type arguments.

C# 14: first-class `Span<T>` conversions

C# 14 adds implicit conversions among `T[]`, `Span<T>`, and `ReadOnlySpan<T>` as first-class language conversions. An array argument can bind directly to a `Span<T>` / `ReadOnlySpan<T>` parameter, spans compose with other conversions, and span types work as extension-method receivers and in generic inference.

```

using System;

static int Sum(ReadOnlySpan<int> s) { int t = 0;
    foreach (var x in s) t += x; return t; }

int[] data = [1, 2, 3, 4];
int total = Sum(data);           // C# 14: T[] -
    > ReadOnlySpan<int> implicitly
Span<int> mutable = data;         // T[] ->
    Span<int> implicitly
ReadOnlySpan<int> ro = mutable;  // Span<T> ->
    ReadOnlySpan<T> implicitly

```

The improvement is partly ergonomic — fewer `.AsSpan()` calls — and partly about generic inference: methods generic over the element type can now infer through these conversions. The first-class span conversions are recognized by the compiler/runtime relationship, so they target `net10.0`. Treat them as the nudge to standardize on span-based APIs: a single `ReadOnlySpan<T>` overload now serves array, span, and `stackalloc` callers without overload duplication.

C# 14: partial constructors and partial events

`partial` extends to instance constructors and events. The rule mirrors partial methods/properties: exactly one *defining* declaration and one *implementing* declaration. For constructors, only the implementing part may carry a `this()` / `base()` initializer, and only one partial declaration may use primary-constructor syntax. For events, the defining part declares a field-like event; the implementing part must supply `add` / `remove`.

```

partial class Widget
{
    public partial Widget(string name);           //
        C# 14: defining partial constructor

    public partial event EventHandler? Clicked;    //
        C# 14: defining partial event
}

partial class Widget
{
    private readonly string _name;
    private EventHandler? _clicked;

    public partial Widget(string name) : this()    //
        implementing part may have an initializer
        => _name = name;

    public Widget() { }

    public partial event EventHandler? Clicked
    {
        add => _clicked += value;                 //
            implementing part supplies add/remove
        remove => _clicked -= value;
    }
}

```

These exist almost entirely for source generators: a generator emits the defining declaration (the public surface) while your hand-written file supplies the body, or vice versa. They complete the “generator declares the API, you implement it” pattern across all member kinds.

C# 14: user-defined compound assignment operators

Historically `a += b` for a user type always lowered to `a = a + b`, allocating a fresh instance from `operator +`. C# 14 lets a type declare an *instance* compound-assignment operator that mutates in place. The declaration uses the `operator` keyword, takes the right-hand operand, has `void` **return**, and carries no `static` modifier — it mutates `this`.

```

class Accumulator
{
    public long Count { get; private set; }

    // C# 14: user-defined instance compound
    // assignment – mutates in place, returns void
    public void operator +=(int amount) => Count +=
        amount;
    public void operator ++() => Count++; //
        user-defined instance increment
}

var acc = new Accumulator();
acc += 10; // calls operator += : no new Accumulator
           // allocated
acc += 5; // Count == 15
acc++; // Count == 16

```

The win is for large or reference-semantics value aggregators (think `BigInteger`-like accumulators, matrix/vector buffers, or pooled builders) where the old `a = a + b` allocation was pure waste. When a type defines *both* `operator +` and an instance `operator +=`, the compound form binds to the in-place `+=`. Do not add these to types whose `+=` semantics the caller expects to be value-copying — silently switching to mutation can surprise code that aliased the value. Reserve them for types that are already mutable and where in-place is the obvious intent.

C# 14: modifiers on simple lambda parameters

You can now put `ref`, `out`, `in`, `ref readonly`, or `scoped` on a lambda parameter *without* writing the parameter type — the type is inferred from the delegate. Previously, any modifier forced you to type every parameter.

```

delegate bool TryParse<T>(string text, out T result);

// C# 14: 'out' modifier without an explicit parameter
// type
TryParse<int> parse = (text, out result) =>
    int.TryParse(text, out result);

bool ok = parse("42", out int n); // ok == true, n ==
42

```

`params` is the lone exception: it still requires an explicitly typed parameter list. This removes a long-standing papercut when writing lambdas for delegates with `out` / `ref` parameters.

Every C# 14 feature in one file

```
// ===== C# 14 (.NET 10) feature roll-up =====
using System;
using System.Collections.Generic;
using System.Linq;

class CSharp14Tour
{
    // field keyword
    public string Name { get; set => field =
        value?.Trim() ?? ""; }

    // user-defined compound assignment + increment
    public int Hits { get; private set; }
    public void operator +=(int n) => Hits += n;
    public void operator ++() => Hits++;

    static void Main()
    {
        var t = new CSharp14Tour();
        t += 3; t++;
        // in-place compound assignment

        // null-conditional assignment (RHS short-
        // circuits if t is null)
        CSharp14Tour? maybe = t;
        maybe?.Name = " trimmed ";
        // -> "trimmed"

        // extension member (property) + first-class
        // span conversion
        int[] xs = [1, 2, 3];
        bool empty = xs.AsEnumerable().IsEmpty;
        // extension property
        int sum = Sum(xs);
        // T[] -> ReadOnlySpan<int>

        // nameof on unbound generic
        Console.WriteLine($"{nameof(List<>)} empty=
            {empty} sum={sum}");

        // simple lambda parameter with a modifier, no
        // explicit type
        Func<string, int> len = s => s.Length;
        TryParse<int> p = (s, out r) =>
            int.TryParse(s, out r);
        _ = (len, p);
    }
}
```

```

static int Sum(ReadOnlySpan<int> s) { int a = 0;
    foreach (var v in s) a += v; return a; }
delegate bool TryParse<T>(string text, out T
    result);
}

static class Ext
{
    extension<T>(IEnumerable<T> src)
    {
        public bool IsEmpty => !src.Any();
        // extension property
    }
}

// partial constructor + partial event in one type
partial class Service
{
    public partial Service(string id);
    public partial event EventHandler? Ready;
}
partial class Service
{
    readonly string _id; EventHandler? _ready;
    public partial Service(string id) => _id = id;
    public partial event EventHandler? Ready { add =>
        _ready += value; remove => _ready -= value; }
}

```

Worked example: tying it together

The following minimal in-memory metrics sink combines five C# 14 features — `field`-backed property, extension property, user-defined `+=`, null-conditional assignment, and span conversions — in code you might plausibly ship.

```

using System;
using System.Collections.Generic;
using System.Linq;

sealed class Counter
{
    // field keyword: validated/normalized, no manual backing field.
    // Normalizing in the setter keeps the backing field non-null (no nullable warnings).
    public string Label
    {
        get => field;
        set => field = value?.Trim() is { Length: > 0 } v ? v : "(unnamed)";
    } = "(unnamed)"; // property initializer seeds the synthesized backing field

    public long Value { get; private set; }

    // user-defined compound assignment: in-place, no reallocation
    public void operator +=(long delta) => Value += delta;
}

static class MetricsExtensions
{
    extension(ReadOnlyList<Counter> counters)
    {
        // extension property over a collection you don't own
        public long Grand => counters.Sum(c => c.Value);
        public bool AnyNonZero => counters.Any(c => c.Value != 0);
    }
}

class Program
{
    static long Tally(ReadOnlySpan<long> samples) // span receiver
    {
        long t = 0;
        foreach (var s in samples) t += s;
        return t;
    }

    static void Main()
    {

```



```

var requests = new Counter { Label = "
http.requests " }; // setter normalizes ->
"http.requests"
requests += 7;           // user-defined +=
requests += 3;           // Value == 10

Counter? optional = DateTime.UtcNow.Ticks % 2
== 0 ? new Counter() : null;
// null-conditional assignment: the setter
// (and its RHS) is skipped entirely
// when 'optional' is null, so this never
// dereferences a null receiver.
optional?.Label = " optional "; // C# 14:
// assigns only if optional is non-null
if (optional is not null) optional += 5; //
// user-defined += needs a non-null receiver

long[] batch = [10, 20, 30];
long total = Tally(batch);           //
long[] -> ReadOnlySpan<long>

var all = new[] { requests }.ToList();
Console.WriteLine($"{requests.Label}:
{requests.Value}, grand={all.Grand}, batch=
{total}");
// http.requests: 10, grand=10, batch=60
    }
}

```

The notable performance characteristic: every modern construct here is allocation-neutral or allocation-reducing. The `field` properties synthesize one field each (identical to hand-written), the extension members are static calls with no instance overhead, `operator +=` mutates rather than allocating, and the span conversion lets `Tally` accept an array, a `stackalloc` span, or a slice with no copying. None of these features introduces hidden boxing, virtual dispatch, or GC pressure — that is by design, and it is why they are safe to adopt broadly in hot paths.

Migration checklist: C# 11/12 → C# 14

Upgrading is low-risk but not zero-risk. Work the list in order:

1. Bump `<TargetFramework>` to `net10.0` first, then `<LangVersion>`. Most C# 14 features (`field`, extension members, null-conditional assignment, `nameof` unbound

generics, lambda modifiers, partial constructors/events) compile against older TFMs, but the two runtime-bound features set framework floors: `System.Threading.Lock` (C# 13) needs `net9.0` +, and **first-class span conversions** (C# 14) effectively need `net10.0` — using C# 14 on a TFM older than `net10.0` is an unsupported configuration. Do not assume “the SDK is enough.”

2. **Audit identifiers named `field`**. Before raising `LangVersion`, grep for any local, parameter, or field literally named `field` that is referenced inside a property accessor. Each becomes ambiguous; qualify with `this.field` / `@field` or rename. This is the only silent behavior change in the set.
3. **Audit object-typed locks**. If you migrate `lock` (`someObject`) to `System.Threading.Lock`, change the *declared type* of the variable to `Lock`. An object-typed variable holding a `Lock` triggers a warning and the slow `Monitor` path.
4. **Re-examine `params T[]` hot paths**. Add `params ReadOnlySpan<T>` overloads (C# 13) to variadic APIs to remove per-call array allocations; keep the array overload for binary compatibility.
5. **Review `a = a + b` patterns on your value-ish types**. Where in-place mutation is correct and allocations are measurable, add C# 14 `operator +=`. Where callers expect copy semantics, leave it alone.
6. **Re-run analyzers and tests**. Enable the latest analyzers; the Roslyn `field` /extension/null-conditional diagnostics surface the migration hazards above. A green build plus a full test pass is the bar before merging the `LangVersion` bump.
7. **Sweep null-conditional assignment opportunities last**. Convert `if (x is not null) x.P = v;` to `x?.P = v;` only after confirming the RHS has no side effects you depend on running unconditionally — the RHS short-circuits with the receiver.

A purely mechanical first pass (steps 1–3) is what makes the upgrade safe; steps 4–7 are opt-in modernizations you can land incrementally.

Quick reference: feature → version → framework

→ chapter

Feature	C#	Needs	Where covered
Primary constructors (all types)	12	net8.0	Ch. 5
Collection expressions <code>[a, b, ..s]</code>	12	net8.0 (spread lowering improves on newer)	Ch. 11
<code>ref readonly</code> parameters	12	net8.0	Ch. 14
Default lambda parameters	12	net8.0	Ch. 9
<code>using</code> alias for any type	12	net8.0	Ch. 3
Inline arrays <code>[InlineArray]</code>	12	net8.0	Ch. 14
<code>params</code> collections (Span/IEnumerable/...)	13	SDK; span form best on net9.0+	Ch. 9, this ch.
<code>System.Threading.Lock</code> + <code>lock</code> pattern	13	net9.0+ (run-time type)	Ch. 19, this ch.
<code>\e</code> escape sequence	13	SDK only	this ch.
<code>ref</code> / <code>unsafe</code> in iterators & async	13	SDK only	Ch. 14, this ch.
<code>allows ref struct</code> + ref struct interfaces	13	SDK only	Ch. 14, this ch.
Implicit <code>^</code> index in initializers	13	SDK only	Ch. 11, this ch.
<code>[OverloadResolutionPriority]</code>	13	SDK only	Ch. 7, this ch.

Feature	C#	Needs	Where covered
Partial properties / indexers	13	SDK only	Ch. 6, this ch.
<code>field</code> keyword	14	SDK only	Ch. 6, this ch.
Extension members (extension blocks)	14	SDK only	Ch. 9, this ch.
Null-conditional assignment	14	SDK only	Ch. 8, this ch.
<code>nameof</code> of unbound generics	14	SDK only	this ch.
First-class <code>Span<T></code> conversions	14	net10.0 (runtime relationship)	Ch. 14, this ch.
Partial constructors & events	14	SDK only	Ch. 6, this ch.
User-defined compound assignment	14	SDK only	Ch. 7, this ch.
Modifiers on simple lambda params	14	SDK only	Ch. 9, this ch.

The two rows in bold are the gates worth memorizing: anything else is purely a compiler concern you can adopt the moment your SDK is current, but `System.Threading.Lock` and the first-class span conversions reach into the runtime and therefore set a target-framework floor.

Appendix · Tooling, Diagnostics, and the Modern Dev Environment

A language is only as sharp as the toolchain around it. This appendix is the field manual for the C# 14 / .NET 10 developer: how to read what the JIT actually emits, how to attach diagnostics to a live process, how to make the compiler enforce modern style as a hard build error, and how to wire a repository so that nullability, package versions, and secrets are governed centrally rather than per-project. Treat it as reference — dense recipes you will copy into real solutions, with the traps that bite people in production called out explicitly.

Reading IL and JIT asm: SharpLab, ILSpy, and the DisassemblyDiagnoser workflow

Every modern C# feature compiles down to IL, and the JIT lowers that IL to machine code. When you reason about a `Span<T>` conversion, a `field`-backed property, or whether a primary-constructor parameter became a hidden field, you are reasoning about a transformation you cannot see in source. Three tools expose successive layers.

SharpLab (sharplab.io) is the fastest loop. Paste C#, pick an output: *C#* (decompiled — see what the compiler lowered your sugar into), *IL*, or *JIT Asm*. It is invaluable for the new C# 14 surface. Drop in a `field`-keyword property (introduced in C# 14) and SharpLab's decompiled-C# view shows the compiler-synthesized `<P>k__BackingField` and the trivial accessor — proof that `field` is pure syntactic sugar with zero runtime cost over a hand-written backing field. Do the same for an `extension` block (C# 14): the decompiled output reveals the static method the extension member lowers to, confirming there is no vtable dispatch and no allocation.

ILSpy (and its CLI `ilspycmd`, plus the VS Code extension) is for whole-assembly archaeology — decompiling a NuGet package

you don't have source for, or confirming what a Release build of *your own* assembly contains after the compiler optimized it. Install the cross-platform CLI as a global tool:

```
// dotnet tool install --global ilspycmd
// ilspycmd MyLib.dll -il           // dump IL to
//                                stdout
// ilspycmd MyLib.dll -p -o ./out   // decompile to a
//                                compilable C# project on disk
// ilspycmd MyLib.dll -l c         // list types
//                                (here: classes; i/s/d/e for
//                                interface/struct/delegate/enum)
```

BenchmarkDotNet's **DisassemblyDiagnoser** is the authoritative source for production asm because it disassembles the *actually-JITted, fully-optimized, tiered-up* code for a benchmark, not a synthetic snippet. The diagnoser (shipped since BenchmarkDotNet 0.10.10) emits ASM, IL, and the originating C# side by side. Attach it with an attribute:


```

using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

[DisassemblyDiagnoser(maxDepth: 2, printSource: true)]
// printSource interleaves C#; maxDepth
// follows callees 2 levels deep
public class SpanSum
{
    private readonly int[] _data =[..
        Enumerable.Range(0, 1024)]; // collection
        expression, C# 12

    [Benchmark(Baseline = true)]
    public int LinqSum() => _data.Sum();

    [Benchmark]
    public int SpanSum_Manual()
    {
        // T[] -> ReadOnlySpan<T> has been an implicit
        // conversion since Span shipped;
        // C# 14 promotes it to a first-class
        // *language* conversion so it also composes
        // with other conversions, works as an
        // extension-method receiver, and participates
        // in generic type inference. The plain local
        // assignment here was always legal.
        ReadOnlySpan<int> span = _data;
        int total = 0;
        for (int i = 0; i < span.Length; i++)
            total += span[i];
        return total;
    }
}

// Entry point – run in RELEASE, never Debug (see
// pitfalls):
// dotnet run -c Release
internal static class Program
{
    private static void Main(string[] args) =>
        BenchmarkRunner.Run<SpanSum>();
}

```

The generated `*-asm.md` report shows the JIT eliminating bounds checks in `SpanSum_Manual` because the loop's upper bound is `span.Length` — a pattern the JIT recognizes and proves safe. Reading that asm is how you *confirm* a bounds-check elimination rather than assuming it. The `maxDepth` para-

meter (default `1`, which disassembles just the benchmark method body, indexed from 1) controls how far into callees the disassembler recurses; raise it to see whether a helper inlined.

A practical workflow: prototype the shape in SharpLab (instant, no project), confirm the lowering is what you expect, then — only if the hot path matters — promote it to a BenchmarkDotNet harness with `DisassemblyDiagnoser` to see the real optimized asm and a measured number. ILSpy is the fallback when the code you care about lives in a third-party DLL.

Diagnostic CLI tools: `dotnet-trace`, `dotnet-counters`, `dotnet-gcdump`, `dotnet-dump`

These four `dotnet-*` global tools attach to a *running* process by PID — no recompilation, no code changes, and they work in containers. Install them once:

```
// dotnet tool install --global dotnet-counters
// dotnet tool install --global dotnet-trace
// dotnet tool install --global dotnet-gcdump
// dotnet tool install --global dotnet-dump
```

Each targets a different question:

- **`dotnet-counters`** — live, low-overhead metrics (the first thing to run). Real-time CPU, allocation rate, GC heap sizes, gen-0/1/2 counts, exception rate, ThreadPool queue length, all via the legacy `EventCounter` and the newer `System.Diagnostics.Metrics` pipelines. Since .NET 9 the runtime exposes a richer `System.Runtime Meter` (GC pause time, IL bytes jitted, methods jitted, CPU time in user/system mode, ThreadPool metrics) over the older `EventCounters`, so the default counter set is fuller than on .NET 8 and earlier — and .NET 10 keeps building on that `Meter`-based surface.
- **`dotnet-trace`** — captures an execution trace (`.nettrace`) of CPU samples, GC events, and contention for offline analy-

sis in PerfView, Visual Studio, or Speedscope. This is your CPU profiler when you cannot attach a GUI.

- `dotnet-gcdump` — a *managed heap* snapshot (`.gcdump`) for finding leaks: which types are rooted, how many instances, retained size. Lightweight; safe-ish on production because it triggers a GC but does not pause for a full process dump.
- `dotnet-dump` — a full process dump plus an SOS-powered analysis REPL (`dumpheap -stat` , `clrstack` , `gcroot`) for post-mortem of hangs, deadlocks, and crashes.

The canonical live session — capture GC and CPU on a running sample app:

```

// 1. Find the PID of the running app (works in a
//     container too):
//     dotnet-counters ps
//
//     14872 SampleApi /app/SampleApi
//
// 2. Live health dashboard – refreshes in place.
//     Watch alloc rate + gen-2 GC count.
//     dotnet-counters monitor --process-id 14872 \
//         --counters
//         System.Runtime,Microsoft.AspNetCore.Hosting
//
//     [System.Runtime]
//         CPU Usage (%)                63
//         GC Heap Size (MB)            412
//         Gen 2 GC Count (Count / 1 interval) 3  <-
//         climbing == promotion pressure / leak
//         Allocation Rate (B / 1 interval) 88,201,344
//         ThreadPool Queue Length      21  <-
//         sustained > 0 == thread starvation
//
// 3. If counters point at GC pressure, capture a 20s
//     CPU+GC trace for offline analysis:
//     dotnet-trace collect --process-id 14872 --
//         duration 00:00:20 \
//         --profile gc-verbose --output api-
//         gc.nettrace
//
// 4. Suspect a managed leak? Snapshot the heap and
//     diff two captures over time:
//     dotnet-gcdump collect --process-id 14872 --
//         output before.gcdump
//     # ...drive load for a few minutes...
//     dotnet-gcdump collect --process-id 14872 --
//         output after.gcdump
//     # Open both in Visual Studio; compare retained
//         sizes by type.
//
// 5. Hard hang / deadlock – full dump + analyze:
//     dotnet-dump collect --process-id 14872 --output
//         hang.dmp
//     dotnet-dump analyze hang.dmp
//     > dumpheap -stat          # type histogram,
//         biggest retained types
//     > threads                 # all managed threads
//     > clrstack -all           # every thread's
//         managed stack -> find the blocked one

```

The decision tree in practice: `dotnet-counters` tells you *whether* there's a problem and roughly *what kind* (CPU? GC?)

thread starvation?). If it's CPU or allocation, escalate to `dotnet-trace`. If the GC heap is monotonically growing, escalate to `dotnet-gcdump` and diff. If the process is wedged or has crashed, go straight to `dotnet-dump`. Counters are cheap enough to leave running in production; full dumps pause the process and should be reserved for genuine incidents.

A small sample app worth instrumenting — it deliberately churns the gen-0 heap so the counters above light up:

```
using System.Diagnostics.Metrics;

// A named Meter (System.Diagnostics.Metrics) surfaces
// a custom counter
// alongside the built-in System.Runtime ones in
// dotnet-counters.
var meter = new Meter("Sample.Work", "1.0");
var processed = meter.CreateCounter<long>
    ("items.processed");

var rng = new Random();
while (true)
{
    // Allocate short-lived buffers on purpose ->
    // visible Allocation Rate + gen-0 churn.
    var buffer = new byte[rng.Next(4_096, 65_536)];
    rng.NextBytes(buffer);
    processed.Add(1);

    if (buffer[0] == 0) await Task.Delay(1); // keep
    // it from pinning a core at 100%
}
// Run, then: dotnet-counters monitor --process-id
// <pid> --counters System.Runtime,Sample.Work
```

EditorConfig, analyzers, and enforcing modern-C# style as build errors

Style consistency that only lives in a formatter is advisory; style that the *build* rejects is policy. .NET ships a large catalog of analyzers in two families: **code-quality** rules (`CAXxxx` — correctness, performance, security) and **code-style** rules (`IDExxxx` — formatting and idiom). The compiler also has **warning waves**: opt-in diagnostics gated by warning level, where wave 10

(`/warn:10`) was added with C# 14. None of these are errors by default. You promote them.

Two MSBuild switches control the baseline. `AnalysisLevel` selects the analyzer/warning-wave generation (`latest` tracks the SDK); `EnforceCodeStyleInBuild` is the crucial one — without it, `IDExxxx` style rules are reported only by the IDE, *not* by `dotnet build` or CI. Turn it on and the same `.editorconfig` severities apply at build time.

An `.editorconfig` snippet that promotes high-value modern-C# diagnostics to error severity (this is per-rule, so you escalate selectively rather than flipping everything):

```

// .editorconfig (NOT C# – shown in a code fence for
// layout only)
// root = true
//
// [*.cs]
// # --- Nullability: make annotation/dereference
// mistakes fail the build ---
// dotnet_diagnostic.CS8600.severity = error //
// converting null literal to non-nullable
// dotnet_diagnostic.CS8602.severity = error //
// dereference of a possibly-null reference
// dotnet_diagnostic.CS8625.severity = error // null
// literal to non-nullable param
//
// # --- Modern-C# idioms as errors (IDE rules; need
// EnforceCodeStyleInBuild=true) ---
// dotnet_diagnostic.IDE0005.severity = error //
// remove unnecessary usings
// dotnet_diagnostic.IDE0011.severity = error //
// require braces on if/else
// dotnet_diagnostic.IDE0090.severity = error // use
// 'new()' target-typed
// dotnet_diagnostic.IDE0300.severity = error // use
// collection expression (C# 12 [..])
// dotnet_diagnostic.IDE0290.severity = error // use
// primary constructor (C# 12)
//
// # --- Style PREFERENCES that the above rules
// enforce ---
// csharp_style_prefer_primary_constructors =
// true:error
// dotnet_style_prefer_collection_expression =
// true:error
// csharp_prefer_braces = true:error
// csharp_style_var_when_type_is_apparent =
// true:suggestion
//
// # --- Performance CA rules worth failing on in
// libraries ---
// dotnet_diagnostic.CA1822.severity = error // mark
// members static where possible
// dotnet_diagnostic.CA1859.severity = warning // use
// concrete types for perf

```

The `IDE0300` / `IDE0290` rules are what make C# 12 collection expressions and primary constructors *mandatory* in your code-base rather than merely permitted — useful when you want a uniform style across a large team. The `true:error` form on the preference line and the `dotnet_diagnostic.<id>.severity = error` form work together: the preference says *what* the id-

onomic shape is, the diagnostic severity says *how hard* to enforce it.

Roslyn analyzer/severity configuration and warning-wave opt-ins

Severity resolution has two layers, and confusing them is what makes people chase a “why is this still a warning” ghost. The first layer decides a diagnostic’s *effective severity*, most-specific-wins: an inline `#pragma warning disable` (lexically scoped, beats everything for the lines it covers), then a per-rule `.editorconfig` entry (`dotnet_diagnostic.<id>.severity`), then `.editorconfig` category/all-rules bulk entries (`dotnet_analyzer_diagnostic...`), then `AnalysisMode` / `AnalysisLevel` defaults, then the rule’s shipped default. The second layer is the MSBuild `<NoWarn>` / `<WarningsAsErrors>` / `<TreatWarningsAsErrors>` properties, which are applied by the compiler *on top of* that effective severity: `<NoWarn>` suppresses an ID outright and `<WarningsAsErrors>` promotes a still-emitted warning to an error — so they can override what `.editorconfig` resolved to. (One catch: enabling rules in bulk via the `<AnalysisMode>` / `<AnalysisLevel>` MSBuild properties causes any bulk `dotnet_analyzer_diagnostic .editorconfig` entries to be ignored; per-rule `dotnet_diagnostic` entries still apply.)

`AnalysisMode` is the blunt instrument that sets the *default* for an entire analyzer category:

```
// In a .csproj or Directory.Build.props:
// <AnalysisLevel>latest</AnalysisLevel>
//     SDK-current analyzers + warning wave
// <AnalysisMode>All</AnalysisMode>
//     enable EVERY CA rule by default (aggressive)
// <AnalysisModeSecurity>All</AnalysisModeSecurity>
//     ...or only escalate one category
// <CodeAnalysisTreatWarningsAsErrors>true</...>
//     CA warnings -> errors without touching CS
//     warnings
```


`AnalysisMode=All` enables analyzers that are off by default; combined with `.editorconfig` you then *down-tune* the handful that are too noisy for your codebase (set them to `silent` or `none`) rather than enabling rules one by one. This “enable everything, then suppress with a documented reason” posture scales better than an allowlist.

Warning waves are version-gated and opt-in. The compiler that ships with .NET 10 (the C# 14 compiler) carries wave-10 diagnostics. You enable them by raising `AnalysisLevel` (or `WarningLevel` directly to 10). Suppressing a *specific* wave warning while keeping the rest is

`<NoWarn>$(NoWarn);CS9XXX</NoWarn>` — additive, so you never clobber the SDK’s existing `NoWarn` list. Escalating one to an error without making all warnings errors is

`<WarningsAsErrors>$(WarningsAsErrors);CS8602</WarningsAsErrors>`.

Project file recipes: a solution-wide `Directory.Build.props`

`Directory.Build.props` is auto-imported by MSBuild into every project below it in the directory tree — the single place to set policy once. This one enforces nullable reference types, treats warnings as errors, and pins analyzer levels solution-wide:

```

<!-- Directory.Build.props at the repository root -->
<Project>
  <PropertyGroup>
    <!-- Language + target. LangVersion
         'latest'/'preview' opt into C# 14 surface. -->
    <TargetFramework>net10.0</TargetFramework>
    <LangVersion>latest</LangVersion>

    <!-- Nullable reference types ON everywhere (C#
         8+; the modern default). -->
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>

    <!-- Make warnings fail the build, EVERYWHERE. See
         the pitfall below before flipping this. -->

    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>

    <!-- Analyzers: current generation + run IDE style
         rules during build. -->
    <AnalysisLevel>latest</AnalysisLevel>
    <AnalysisMode>Recommended</AnalysisMode>

    <EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>
    <EnableNETAnalyzers>true</EnableNETAnalyzers>

    <!-- Deterministic, debuggable Release builds (see
         Source Link section). -->
    <Deterministic>true</Deterministic>
    <ContinuousIntegrationBuild Condition="'$(CI)' ==
         'true'">true</ContinuousIntegrationBuild>

    <!-- Generate XML docs so CS1591 (missing doc)
         participates – drop for app projects. -->

    <GenerateDocumentationFile>true</GenerateDocumentationFile>

    <!-- Escape hatch: a few warnings that are noise
         during migration. Additive $(NoWarn)). -->
    <NoWarn>$(NoWarn);CS1591</NoWarn>
  </PropertyGroup>
</Project>

```

For libraries shipped as NuGet, or apps using **Native AOT**, the relevant trimming/AOT switches belong in the *project* (or a conditioned group) because they change the build profoundly:

```

<!-- In an AOT-published executable's .csproj -->
<PropertyGroup>
  <PublishAot>true</PublishAot> <!--
    Native AOT: ahead-of-time, no JIT, single
    native binary -->

    <InvariantGlobalization>true</InvariantGlobali-
    zation> <!-- drop ICU; smaller, faster startup
    -->

  <StackTraceSupport>false</StackTraceSupport> <!--
    shave size if you don't need pretty traces -->
</PropertyGroup>

<!-- In a trimmed (but not AOT) library that wants
  trim-safety verified at build -->
<PropertyGroup>
  <IsTrimmable>true</IsTrimmable> <!--
    mark assembly trimmable; enables IL2xxx
    analyzers -->
  <EnableTrimAnalyzer>true</EnableTrimAnalyzer> <!--
    surface reflection/trim hazards as warnings --
    >
</PropertyGroup>

```

`PublishAot` implicitly enables the trim and AOT analyzers, which will warn (`IL2xxx` / `IL3xxx`) on reflection patterns that can't survive trimming — invaluable for catching `Activator.CreateInstance` and `JsonSerializer` reflection paths before they fail at runtime as a missing type. With `TreatWarningsAsErrors=true` from the props above, those become build failures, which is exactly what you want for an AOT target.

Central package management:

Directory.Packages.props

Central Package Management (CPM) moves every version number into one file at the repo root. Enable it by setting `ManagePackageVersionsCentrally`, declare each version once with `<PackageVersion>`, and then `<PackageReference>` entries in project files carry `no Version`:

```

<!-- Directory.Packages.props at the repository root -
-->
<Project>
  <PropertyGroup>

    <ManagePackageVersionsCentrally>true</ManagePa
ckageVersionsCentrally>

    <!-- Fail the build if any project still pins a
version inline -> drift is impossible. -->

    <CentralPackageTransitivePinningEnabled>true</
CentralPackageTransitivePinningEnabled>
  </PropertyGroup>

  <ItemGroup>
    <!-- One source of truth for every package version
in the solution. -->
    <PackageVersion
      Include="Microsoft.Extensions.Hosting"
      Version="10.0.0" />
    <PackageVersion
      Include="Microsoft.Extensions.Configuration"
      Version="10.0.0" />
    <PackageVersion Include="System.Text.Json"
      Version="10.0.0" />
    <PackageVersion Include="BenchmarkDotNet"
      Version="0.15.0" />
    <PackageVersion Include="xunit.v3" Version="3.2.2"
      />    <!-- the v3 package line is versioned
3.x, NOT 2.x -->
    <PackageVersion Include="Microsoft.NET.Test.Sdk"
      Version="17.13.0" />
  </ItemGroup>

  <ItemGroup>
    <!-- GlobalPackageReference: injected into EVERY
project automatically (analyzers, SourceLink).
-->
    <GlobalPackageReference
      Include="Microsoft.SourceLink.GitHub"
      Version="8.0.0" />
  </ItemGroup>
</Project>

```

And a consuming project — note the bare references:

```

<!-- src/SampleApi/SampleApi.csproj -->
<Project Sdk="Microsoft.NET.Sdk.Web">
  <ItemGroup>
    <PackageReference
      Include="Microsoft.Extensions.Hosting" /> <!--
      version resolved centrally -->
    <PackageReference Include="System.Text.Json" />
  </ItemGroup>
</Project>

```

The payoff: a single PR bumps a transitive vulnerable dependency for the whole solution, Dependabot has one file to edit, and `CentralPackageTransitivePinningEnabled` lets you pin a *transitive* dependency's version (to dodge a CVE in something you don't reference directly) without adding a phantom direct reference. `GlobalPackageReference` is the right vehicle for cross-cutting build-time packages — analyzers, Source Link — so individual projects never have to remember them.

Doppler-style secret handling and config in sample apps

Secrets do not belong in source, in `appsettings.json`, or in `.csproj`. The .NET configuration system layers providers in order, and the *last* provider wins, so environment-injected secrets cleanly override file defaults. A Doppler-style workflow injects secrets as environment variables at process start (`doppler run -- dotnet run`), and the app reads them through the standard `IConfiguration` chain — no SDK dependency, no provider-specific code:

```

using Microsoft.Extensions.Configuration;

var config = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json", optional: false)
    // committed: structure + safe defaults
    .AddJsonFile($"appsettings.
    {Environment.GetEnvironmentVariable("DOTNET_ENVIRONMENT")}.json",
    optional: true)
    // committed: per-env non-secret overrides
    .AddUserSecrets<Program>(optional: true)
    // LOCAL ONLY: ~/.microsoft/usersecrets, never
    committed
    .AddEnvironmentVariables()
    // LAST = highest precedence; Doppler injects
    here
    .Build();

// Hierarchical keys map from env vars via '___':
// ConnectionStrings__Default ->
// ConnectionStrings:Default
string conn = config.GetConnectionString("Default")
    ?? throw new InvalidOperationException(
        "ConnectionStrings__Default not
        set. Run via: doppler run -- dotnet run");

string apiKey = config["Clarqo:ApiKey"]
    // env var Clarqo__ApiKey (keys are case-
    insensitive;

    // '___' -> ':', so it is ApiKey, NOT API_KEY)
    ?? throw new
    InvalidOperationException("Clarqo:ApiKey
    missing from environment.");

// appsettings.json carries ONLY structure +
// placeholders, e.g.:
// { "ConnectionStrings": { "Default": "" },
//   "Clarqo": { "ApiKey": "" } }
// The real values arrive via the environment provider
// - the file is safe to commit.

```

The discipline that makes this safe: the committed JSON contains the *shape* (so config binding and the options pattern work) but empty or non-sensitive values; user-secrets cover local dev without ever touching the repo; and the environment provider — fed by Doppler, Key Vault, or your platform’s secret store in CI/prod — supplies the real values and wins by ordering. The `___` double-underscore convention is how a flat environment

variable maps onto a hierarchical config key, which is what lets a secret manager that only speaks env vars drive deeply-nested .NET configuration.

Symbol servers, Source Link, and debugging into the BCL

To step into `List<T>.Add` or `string.Concat` during a debug session, the debugger needs two things: **symbols** (PDBs mapping IL offsets to source lines) and the **source** itself. Symbol servers serve the former; **Source Link** maps PDB entries to a URL where the latter lives.

Configure the debugger to fetch Microsoft and NuGet.org symbols, and enable Source Link and “step into framework code.” For your own NuGet packages, embed the metadata so *consumers* can debug into your library:

```
<!-- In a library's .csproj (or Directory.Build.props
      for all packable projects) -->
<PropertyGroup>
  <PublishRepositoryUrl>true</PublishRepositoryUrl>
  <!-- embed repo URL in the PDB -->
  <EmbedUntrackedSources>true</EmbedUntrackedSources>
  <!-- include generated/untracked source -->
  <IncludeSymbols>true</IncludeSymbols>
  <SymbolPackageFormat>snupkg</SymbolPackageFormat>
  <!-- modern symbol package -> NuGet.org symbol
      server -->
  <DebugType>portable</DebugType>
  <!-- portable PDBs, required for Source Link -
      -->
  <!-- Gate on CI: normalizing source paths breaks
      local source debugging, so only force it
      on the CI build that actually produces the
      published package. -->
  <ContinuousIntegrationBuild Condition="'$(CI)' ==
      'true'">true</ContinuousIntegrationBuild>
</PropertyGroup>
<!-- Plus the Microsoft.SourceLink.GitHub
      GlobalPackageReference from
      Directory.Packages.props above. -->
```

With this, a consumer who hits a bug in your library can set a breakpoint inside it and step through *your actual source* fetched

from GitHub by commit hash — no local clone, guaranteed to match the binary because the PDB pins the exact commit. The same mechanism lets you step into the BCL: the .NET runtime ships Source Link-enabled symbols, so with framework stepping enabled and symbol servers configured, a breakpoint in your code can step down into `System.Text.Json`’s real source. `ContinuousIntegrationBuild=true` matters here — it normalizes file paths in the PDB so the Source Link mapping resolves identically on every machine.

A worked example tying it together

A realistic scenario: a JSON-shaped hot path in a service is allocating more than it should. We profile it, read the asm, and verify a fix — exercising counters, the JIT lowering, and a C# 14 feature in one loop.


```

using System.Diagnostics.Metrics;
using System.Text;
using BenchmarkDotNet.Attributes;

[MemoryDiagnoser] // reports
                  allocated bytes/op – the number we care about
[DisassemblyDiagnoser(maxDepth: 1)] // and the
                  optimized asm for each method
public class TagJoinBenchmark
{
    private readonly string[] _tags = ["ai",
        "finance", "tech", "markets", "gpu"];

    // ALLOCATING baseline: string.Join produces one
    // intermediate string, then the
    // "#" + ... concat allocates a SECOND string. Two
    // heap strings per call.
    [Benchmark(Baseline = true)]
    public string JoinNaive() => "#" + string.Join(
        "#", _tags);

    // Single-pass build: one StringBuilder (object +
    // internal char buffer, pre-sized so it
    // never grows) and one final string from
    // ToString(). The win is avoiding the separate
    // join-then-concat intermediate, not the
    // StringBuilder itself – read the asm/alloc
    // column
    // to see exactly what each costs rather than
    // assuming.
    [Benchmark]
    public string JoinSpan()
    {
        var sb = new StringBuilder(capacity: 64);
        ReadOnlySpan<string> tags = _tags; //
        array -> ReadOnlySpan, a long-standing
        implicit conversion
        for (int i = 0; i < tags.Length; i++)
        {
            if (i > 0) sb.Append(' ');
            sb.Append('#').Append(tags[i]);
        }
        return sb.ToString();
    }
}

// A field-keyword property (C# 14) instrumented with
// a Meter, modeling the service that calls the
// above.
public sealed class TagFormatter
{

```

```

private static readonly Meter Meter =
    new("Sample.Tags", "1.0");
private static readonly Counter<long> Formatted =
    Meter.CreateCounter<long>("tags.formatted");

// C# 14 'field' keyword: a validated property
// with NO explicitly declared backing field.
public int MaxTags
{
    get;
    set => field = value < 1 ? 1 : value;    //
    // 'field' is the compiler-synthesized backing
    // store
} = 16;

public string Format(string[] tags)
{
    Formatted.Add(1);
    int count = Math.Min(tags.Length, MaxTags);
    // .NET 9 added string.Join(string?,
    // ReadOnlySpan<string?>). Its element type is
    // string? (nullable), and ReadOnlySpan<T> is
    // NOT nullability-covariant, so a
    // ReadOnlySpan<string> argument warns CS8620
    // under Nullable=enable. Build the span
    // with the exact element type (string?) so
    // the slice binds to the span overload
    // without the variance warning.
    ReadOnlySpan<string?> span = new
    ReadOnlySpan<string?>(tags, 0, count);
    return "#" + string.Join("#", span);
}
}

```

Running this under `dotnet run -c Release`, the `[MemoryDiagnoser]` column lets you compare the two paths byte-for-byte: `JoinNaive` allocates two strings (the join result plus the `"#" + ...` concat), while `JoinSpan` trades the second intermediate string for a pre-sized `StringBuilder` and a single `ToString()` — the point of the exercise is that you *read the measured number* instead of guessing which is cheaper, because the `StringBuilder` is not free either. The `DisassemblyDiagnoser` asm shows whether the `span[i]` indexing in the loop has its bounds checks elided (the loop's upper bound is `tags.Length`, the pattern the JIT proves safe). In a live service you'd have spotted the original allocation pressure first with `dotnet-counters monitor --counters Sample.Tags, System.Runtime` (watching

`Allocation Rate` and the custom `tags.formatted` counter), then reached for the benchmark to validate the fix. The `field` keyword and the array-to-span conversion are pure ergonomics here — neither changes the IL's performance character, which is precisely why you confirm that with the disassembler rather than assuming it.

Pitfalls

Enabling `TreatWarningsAsErrors` late drowns you. Flip it on a mature codebase and the build erupts with hundreds of nullable (`CS86xx`) and, on AOT/trim targets, `IL2xxx` warnings simultaneously — an unmergeable wall. The migration path is incremental: enable `<Nullable>enable</Nullable>` first and let warnings *stay warnings* while you annotate; or scope nullability with `#nullable enable` per file and expand outward. Only once warnings are at zero do you set

`TreatWarningsAsErrors=true` to *keep* them there. For warning waves, raise `AnalysisLevel` one generation at a time. The goal is a clean baseline you then defend, not a heroic single PR that touches every file.

Profiling a Debug build and trusting the numbers.

Debug build disables JIT optimizations: no inlining, no bounds-check elimination, locals not enregistered, every `[MethodImpl(AggressiveInlining)]` ignored. Asm read from a Debug build bears no resemblance to production, and a microbenchmark of Debug code measures the *absence* of the optimizer. BenchmarkDotNet actively refuses to run against a Debug build for this reason, and `dotnet-trace` / `dotnet-counters` numbers from a Debug process tell you about un-optimized code nobody ships. Always profile and disassemble `-c Release`. The corollary: `DisassemblyDiagnoser` output is only meaningful from Release, after tiered compilation has tiered *up* — which is exactly the JITted code the diagnoser captures, and exactly why it's the authoritative asm source over a Debug SharpLab paste.

Glossary

- **AOT (Ahead-of-Time)** — compiling to native code at publish time (`PublishAot`), eliminating the JIT and producing a self-contained native binary with fast startup and a small footprint, at the cost of no runtime code generation.
- **Analyzer** — a Roslyn component that inspects source during compilation and emits diagnostics (`CAXxxx` quality, `IDExxxx` style, `IL2xxx` trim).
- **Central Package Management (CPM)** — declaring all NuGet versions once in `Directory.Packages.props` via `ManagePackageVersionsCentrally`.
- **Diagnoser** — a BenchmarkDotNet plug-in (`MemoryDiagnoser` , `DisassemblyDiagnoser`) that augments a benchmark run with extra data.
- **field keyword** — C# 14 contextual keyword giving access to a property's compiler-generated backing field without declaring one.
- **gdump** — a managed-heap snapshot for leak analysis; lighter than a full process dump.
- **IL (Intermediate Language)** — the stack-based bytecode the C# compiler emits, which the JIT (or AOT compiler) lowers to machine code.
- **JIT (Just-In-Time)** — the runtime compiler turning IL into optimized machine code, with tiered compilation promoting hot methods.
- **Source Link** — PDB metadata mapping compiled code to a source URL, enabling step-into debugging of NuGet packages and the BCL.
- **Warning wave** — a version-gated set of opt-in compiler warnings; wave 10 ships with C# 14, enabled via `AnalysisLevel` / `WarningLevel`.

Further reading, keyed to the book's parts

- **Part I–II (Language fundamentals, types):** SharpLab for lowering intuition; the C# language-version feature docs

on Microsoft Learn to map syntax to its introducing version (C# 12 collection expressions and primary constructors; C# 13 `params` collections and `Lock`; C# 14 `field`, extension blocks, implicit spans).

- **Part III–IV (Generics, async, spans):** BenchmarkDotNet’s `DisassemblyDiagnoser` and `MemoryDiagnoser` docs; the `System.Memory / Span<T>` performance guidance — the asm-reading workflow in this appendix is the verification loop for those chapters.
- **Part V (Concurrency):** `dotnet-trace` with the contention profile and `dotnet-dump`’s `clrstack -all` for deadlock post-mortems; the .NET threading and ThreadPool counters surfaced by `dotnet-counters`.
- **Part VI (Performance, GC):** `dotnet-counters` GC counters, `dotnet-gcdump` diffing, and the .NET GC configuration docs; the AOT/trim analyzer references for size- and startup-sensitive work.
- **Part VII (Interop, AOT, libraries):** the trimming and Native AOT compatibility guides, Source Link configuration, and the NuGet Central Package Management documentation — everything in the “Project file recipes” and “Source Link” sections here is the operational companion to those chapters.
- **Part VIII (Reference):** the .NET diagnostics tools overview and the code-analysis configuration reference on Microsoft Learn anchor every recipe in this appendix.