C# Cheat Sheet — C# 14 / .NET 10

Targets **C# 14 on .NET 10 (LTS)** · **pink** marks C# 12–14 additions · learn.microsoft.com/dotnet/csharp

PROGRAM & TYPES

```
// Program.cs — top-level statements
using System;
Console.WriteLine($"hi {args.Length}");
return 0;
namespace App.Core;              // file-scoped
global using System.Linq;        // whole assembly
using Sb = System.Text.StringBuilder;
using static System.Math;
using Pt = (double X, double Y); // C#12
```

class	reference type, heap, nullable
struct	value type, copied on assign
record	value equality + with
interface	contract; default members
enum	named integral constants
delegate	typed method reference

BUILT-IN TYPES & LITERALS

sbyte byte	8-bit ±/+
short ushort	16-bit
int uint	32-bit
long ulong	64-bit
nint nuint	pointer-sized
float double	IEEE 754 32/64
decimal	128-bit, money
bool char	true/false, UTF-16
string object	immutable, root

```
var i = 1_000_000; var h = 0xFF; var b = 0b1010;
var d = 3.14; var f = 1.5f; var m = 9.99m;
var u = 3u; var l = 7L; var c = '\u00e9';
const int Max = 10;      // compile time
readonly Guid _id;       // set in ctor
static readonly int[] T = [1,2];
```

STRINGS

```
var s = $"{name,-10} {value:F2} {x:X}";
var v = @"C:\path\no\escapes";
var r = """               // raw, C#11
  {"json": "no escaping"}
  """;
var u8 = "bytes"u8;       // ReadOnlySpan<byte>
s.Length  s[0]  s[^1]  s[1..4]
s.Trim() .ToUpperInvariant() .Split(',')
s.Contains/StartsWith/IndexOf/Replace
string.IsNullOrWhiteSpace(s)
string.Join(", ", items)  string.Concat(a,b)
new StringBuilder().Append(x).ToString()
int.TryParse(s, out var n)
value.ToString("N2", CultureInfo.InvariantCulture)
```

OPERATORS

?? ??=	null-coalesce / assign
?. ?[]	null-conditional
a?.P = v	**C#14** null-cond. assign
^n ..	index-from-end, range
is / is not	pattern test
as	cast or null
=>	lambda / expr body
?:	conditional
&& || !	short-circuit logic
& | ^ ~ << >> >>>	bitwise, unsigned shift
checked/unchecked	overflow behaviour
nameof typeof sizeof	compile-time info
with	non-destructive copy

Precedence: unary > * / > + - > shift > relational > equality > & > ^ > | > && > || > ?? > ?: > assignment.

CONTROL FLOW

```
if (x > 0) { } else if (x < 0) { } else { }
while (c) { } do { } while (c);
for (int i = 0; i < n; i++) { }
foreach (var item in items) { }
await foreach (var x in stream) { }
break; continue; return v; goto label;

switch (code) {                  // statement
    case 1 or 2: Do(); break;
    case int n when n > 9: Big(); break;
    default: break;
}
var text = code switch {         // expression
    0 => "zero",
    > 0 and < 10 => "small",
    _ => "other",
};
```

METHODS & PARAMETERS

```
public static int Add(int a, int b = 0) => a + b;
Add(b: 2, a: 1);                 // named args

void M(ref int r,    // in+out, must be set
       out int o,    // must assign inside
       in int i,     // readonly by ref
       params int[] rest) { o = 0; }

int Sum(params ReadOnlySpan<int> v) => 0; // C#13

static T Local<T>(T x) => x;   // local function
ref int First(int[] a) => ref a[0];
```

CLASS / STRUCT / RECORD

```
public sealed class Account(string id) // primary ctor C#12
{
    private readonly string _id = id;
    public static int Count { get; private set; }

    public Account() : this("anon") { }
    public override string ToString() => _id;
}

public readonly struct Point(double x, double y)
{ public double X { get; } = x;
  public double Y { get; } = y; }

public record Person(string Name, int Age);
public readonly record struct Rgb(byte R,byte G,byte B);
var older = person with { Age = 41 };
var (name, age) = person;        // deconstruct
```

Access: public · internal (assembly) · protected · private protected · protected internal · private (default).

PROPERTIES

```
public string Name { get; set; } = "";
public int Area => W * H;          // computed
public string Id { get; init; }    // set once
public required string Key { get; init; } // C#11

public string Trimmed            // field keyword, C#14
{
    get;
    set => field = value?.Trim() ?? "";
}
public int this[int i] => _items[i];  // indexer
new Person { Name = "Ada", Age = 36 };  // init
```

INHERITANCE & INTERFACES

```
public abstract class Shape : IComparable<Shape>
{
    public abstract double Area { get; }
    public virtual string Label => "shape";
    public int CompareTo(Shape? o) => 0;
}
public sealed class Disc(double r) : Shape
{
    public override double Area => Math.PI*r*r;
    public override string Label => "disc";
}
public interface ILog {
    void Write(string m);
    void Warn(string m) => Write("WARN "+m); // default
    static abstract ILog Create();     // generic math
}
base.M();  obj is Disc d;  obj as Disc;
```

GENERICS

```
public class Cache<TKey, TValue>
    where TKey : notnull, IEquatable<TKey>
    where TValue : class, new() { }

static T Max<T>(T a, T b) where T : INumber<T>
    => a > b ? a : b;              // generic math
```

where T : struct	non-nullable value type
where T : class	reference type
where T : notnull	never null
where T : unmanaged	blittable value type
where T : new()	public parameterless ctor
where T : Base, IFoo	base type / interface
allows ref struct	**C#13** accepts Span<T>
out T / in T	co- / contravariance

PATTERN MATCHING

```
o is null / is not null
o is int n                       // type + capture
o is >= 1 and <= 9               // relational+logical
o is 1 or 2 or 3
p is { Age: > 17, Name.Length: > 0 } // property
p is Person(var nm, var ag)      // positional
arr is [1, .., var last]         // list, C#11
t is (> 0, < 0)                  // tuple

var kind = shape switch {
    Disc { Area: > 100 } => "big disc",
    Disc => "disc",
    null => "none",
    _ => "other",
};
```

DELEGATES, LAMBDAS, EVENTS

```
Func<int,int,int> add = (a,b) => a+b;
Action<string> log = Console.WriteLine;
Predicate<int> even = static n => n%2==0;
Func<int,int> sq = static (int x) => { return x*x; };
delegate int Op(int a, int b);

public event EventHandler<EventArgs>? Done;
Done?.Invoke(this, EventArgs.Empty);
src.Done += OnDone;  src.Done -= OnDone;
```

A lambda capturing a variable allocates a closure; static lambdas capture nothing.

EXTENSION MEMBERS

```
public static class Ext
{
    // classic extension method
    public static bool IsBlank(this string? s)
        => string.IsNullOrWhiteSpace(s);

    extension<T>(IEnumerable<T> src)   // C#14
    {
        public bool IsEmpty => !src.Any();
        public T? Second() =>
src.Skip(1).FirstOrDefault();
    }
}
```

COLLECTION EXPRESSIONS & COLLECTIONS

```
int[] a = [1, 2, 3];                 // C#12
List<int> l = [..a, 4, 5];           // spread
Span<int> s = [0, 0];  int[] e = [];
```

List<T>	growable array; index O(1)
T[]	fixed length; [,] [][]
Dictionary<K,V>	hash map; TryGetValue
HashSet<T>	unique set
Queue/Stack<T>	FIFO / LIFO
SortedDictionary	ordered by key
LinkedList<T>	doubly linked
IReadOnlyList<T>	preferred return type
ImmutableArray<T>	immutable, safe to share
FrozenDictionary	read-optimised, built once

```
IEnumerable<int> Evens() { yield return 2; }
```

LINQ

```
// query syntax
var q = from o in orders
        where o.Total > 100
        orderby o.Date descending, o.Id
        group o by o.Customer into g
        select new { g.Key, Sum = g.Sum(x =>
x.Total) };

// method syntax (equivalent)
orders.Where(o => o.Total > 100)
      .OrderByDescending(o => o.Date).ThenBy(o =>
o.Id)
      .GroupBy(o => o.Customer)
      .Select(g => (g.Key, g.Sum(x => x.Total)));
```

Where Select SelectMany	filter, project, flatten
OrderBy ThenBy Reverse	ordering
GroupBy Join GroupJoin	grouping, joining
Distinct Union Intersect Except	set ops
Take Skip Chunk TakeWhile	partitioning
Any All Contains	quantifiers
Count Sum Min Max Average Aggregate	aggregation
First(OrDefault) Single ElementAt	elements
ToList ToArray ToDictionary ToLookup	materialise

Deferred until enumerated; To* forces it. IQueryable<T> builds an expression tree for a provider (EF Core). Enumerate once.

NULLABILITY

```
// <Nullable>enable</Nullable>
string  notNull = "x";     // warn if null
string? maybe = null;
int?    n = null;          // Nullable<int>
n.HasValue  n.Value  n ?? 0  n.GetValueOrDefault()

maybe?.Length              // null-conditional
maybe ??= "fallback";
var len = maybe!.Length;   // null-forgiving (!)
if (maybe is not null) { }   // flow analysis
ArgumentNullException.ThrowIfNull(arg);
// #nullable enable|disable|restore
```

EXCEPTIONS

```
try {
    Risky();
} catch (IOException ex) when (ex.HResult != 0) {
    Log(ex);  throw;             // keeps stack trace
} catch (Exception ex) {
    throw new InvalidOperationException("ctx",
ex);
} finally {
    Cleanup();                   // always runs
}
var v = maybe ?? throw new ArgumentException("x");
ArgumentException.ThrowIfNullOrWhiteSpace(s);
ArgumentOutOfRangeException.ThrowIfNegative(i);
ObjectDisposedException.ThrowIf(_disposed, this);
```

Never throw ex; (resets the trace). Never catch what you cannot handle. ArgumentException · InvalidOperationException · NotSupportedException · FormatException · KeyNotFoundException.

USING & IDISPOSABLE

```
using var reader = new StreamReader(path); // C#8
using (var conn = Open()) { }        // scoped
await using var stream = OpenAsync();

public sealed class Res : IDisposable
{
    private bool _disposed;
    public void Dispose()
    {
        if (_disposed) return;
        _inner.Dispose();
        _disposed = true;
    }
}
public ValueTask DisposeAsync() =>
_s.DisposeAsync();
```

ASYNC / AWAIT

```
public async Task<string> LoadAsync(
    string url, CancellationToken ct = default)
{
    var body = await _http.GetStringAsync(url, ct)
                          .ConfigureAwait(false);
    return body;
}
var all  = await Task.WhenAll(t1, t2);
var won  = await Task.WhenAny(t1, t2);
await foreach (var t in Task.WhenEach(tasks)) { }
await Task.Delay(100, ct);
await work.WaitAsync(TimeSpan.FromSeconds(5));
using var cts = new CancellationTokenSource(5000);
ct.ThrowIfCancellationRequested();

async IAsyncEnumerable<int> Stream() { yield
return 1; }
```

Async all the way: never .Result/.Wait(). async void only for event handlers. ValueTask<T> awaits once only.

THREADS & parallel

```
lock (_gate) { _count++; }       // _gate: private readonly
private readonly Lock _gate = new();   // .NET 9+
Interlocked.Increment(ref _n);
Interlocked.CompareExchange(ref x, 1, 0);
await _sem.WaitAsync(ct); try { } finally {
_sem.Release(); }
volatile bool _stop;

await Task.Run(() => Heavy(), ct);
Parallel.For(0, n, i => Work(i));
await Parallel.ForEachAsync(urls, opts,
FetchAsync);
data.AsParallel().Where(Heavy).ToArray();
var ch = Channel.CreateBounded<int>(100);
new ConcurrentDictionary<string,int>().GetOrAdd(k,
f);
using var timer = new
PeriodicTimer(TimeSpan.FromMinutes(1));
```

SPAN, MEMORY, STACKALLOC

```
Span<byte> buf = stackalloc byte[256];  // never in a loop
ReadOnlySpan<char> view = text.AsSpan(1, 4);
int year = int.Parse(text[..4]);      // no allocation
span.Slice(2) span[1..3] span.CopyTo(dst)
span.Fill(0)
Memory<byte> mem = array;   mem.Span;  // across await
var pool = ArrayPool<byte>.Shared;
var rented = pool.Rent(8192); try { } finally {
pool.Return(rented); }
```

Span<T> is a ref struct: no fields in classes, no boxing, no lambda capture, never across await. C# 14 makes array↔span conversions implicit and standard.

ATTRIBUTES & REFLECTION

```
[Obsolete("Use X.")] [Flags] [Serializable]
[Conditional("DEBUG")] [ModuleInitializer]
[CallerMemberName]
[CallerArgumentExpression(nameof(c))]
[NotNullWhen(true)] [MemberNotNull(nameof(_x))]
[assembly: InternalsVisibleTo("App.Tests")]

Type t = typeof(Person);   Type rt =
obj.GetType();
t.GetProperties() / .GetMethod("M") /
.GetCustomAttribute<A>()
Activator.CreateInstance(t);  mi.Invoke(obj,
args);
mi.CreateDelegate<Func<int,int>>();   // fast path
typeof(List<>).MakeGenericType(t);
```

Reflection is slow and breaks trimming/AOT — prefer source generators ([GeneratedRegex], [LibraryImport], JsonSerializerContext).

DOTNET CLI

dotnet new console|classlib|xunit|webapi -o N	scaffold
dotnet new sln · dotnet sln add P	solution
dotnet add package P --version V	NuGet
dotnet add reference ../P/P.csproj	project ref
dotnet restore · build -c Release	build
dotnet run -- args	run
dotnet run app.cs	**C#14** file-based app
dotnet watch run|test	hot reload
dotnet test --filter N dotnet test --collect:"XPlat Code Coverage"	test
dotnet format --verify-no-changes	style gate
dotnet publish -c Release -r linux-x64 --self-contained	deploy
dotnet pack · nuget push	package
dotnet tool install -g dotnet-counters	tools
dotnet --info · sdk check	diagnose

```
<!-- csproj essentials -->
<TargetFramework>net10.0</TargetFramework>
<LangVersion>14.0</LangVersion>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
<TreatWarningsAsErrors>true</TreatWarningsAsErrors
>
<GenerateDocumentationFile>true</GenerateDocumenta
tionFile>
```

TESTING (XUNIT)

```
[Fact]
public void Add_WithItem_IncreasesCount()
{
    var b = new Basket();      // arrange
    b.Add("apple");            // act
    Assert.Equal(1, b.Count);  // assert
}
[Theory] [InlineData(2, 4)] [InlineData(3, 9)]
public void Square(int i, int e) =>
Assert.Equal(e, i*i);

Assert.True/False/Null/NotNull/Same/Contains/Empty
Assert.Throws<T>(Act);  await
Assert.ThrowsAsync<T>(F);
// ctor = setup, Dispose = teardown,
IClassFixture<T> = shared
```

NAMING & IDIOMS

PascalCase	types, methods, properties, events, consts
camelCase	parameters, locals
_camelCase	private fields
IName	interfaces
NameAsync	async methods
T, TKey	type parameters

Guard clauses first · readonly and sealed by default · is null over == null · expression bodies for one-liners · var when the type is apparent · return empty, never null · one public type per file · four-space indent, Allman braces.