C# CODING HELP - COMPLETE REFERENCE & PRACTICAL GUIDE
Date: 2026-02-03
Author: Generated for River920
Scope: A consolidated, layered guide with practical examples and best practices covering basics → beginner → mediocre → normal → expert topics.
Note: "Mediocre" here interpreted as lower-intermediate; sections are progressive. This is a comprehensive guide but not literally every conceivable detail (that would be a book). Ask for deeper expansions on any topic.

TABLE OF CONTENTS

1. QUICK CHEAT SHEET
- File extension: .cs
- Entry point (console): static void Main(string[] args) or static Task Main(string[] args)
- Compile/run: dotnet new console; dotnet run
- Targeting: .NET versions (Framework, Core, .NET 5/6/7/8+)
- Access modifiers: public, private, protected, internal, protected internal, private protected
- Common types: int, long, float, double, decimal, bool, char, string, object, dynamic
- Nullable reference types: T? and enable with nullable context
- Value vs Reference: structs = value types, classes = reference types
- Exceptions: throw new Exception("msg"); catch(Exception ex) {}
- Async: async Task/Task<T>/ValueTask<T>, await
- Generics: class Box<T> { T Value; }
- Delegates: delegate int MyDel(string s); Action, Func<T>, Predicate<T>
- LINQ: .Where(...).Select(...).GroupBy(...).OrderBy(...).ToList()
- DI: use Microsoft.Extensions.DependencyInjection
- Testing: xUnit, NUnit, MSTest
- Formatting: use dotnet format or editorconfig
- Analyze: dotnet analyzers / Roslyn analyzers

2. ENVIRONMENT & TOOLING (SETUP)
- Install .NET SDK: https://dotnet.microsoft.com/download
- Editors: Visual Studio (Windows, full-featured), Visual Studio Code (+ C#

extension), Rider (JetBrains)
- CLI essentials:
  - dotnet new console -n MyApp
  - dotnet build
  - dotnet run
  - dotnet test
  - dotnet add package <PackageName>
  - dotnet publish -c Release -r <runtime>
- Project files: .csproj XML format. Example:
  <Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
      <OutputType>Exe</OutputType>
      <TargetFramework>net8.0</TargetFramework>
    </PropertyGroup>
  </Project>
- SDK-style projects support PackageReference, multi-targeting, implicit usings (C# 10+), nullable context.

3. BASICS
3.1 Syntax & Structure
- Namespaces, classes, methods:
  using System;
  namespace MyApp {
    public class Program {
      public static void Main(string[] args) {
        Console.WriteLine("Hello");
      }
    }
  }
- Type inference: var x = 5; // compile-time typed as int
- Constants & readonly:
  const int Max = 10; // compile-time constant
  readonly int _x; // runtime constant (set in ctor)

3.2 Types & Variables
- Value types: int, long, float, double, decimal, bool, char, structs, enums
- Reference types: string, class, delegate, arrays, dynamic, object
- Nullable value types: int? x = null;
- Nullable reference types: enable with <Nullable>enable</Nullable> or #nullable enable ; string? maybe = null;

3.3 Control Flow
- if, else, switch (switch expressions since C# 8), ternary, while, do-while, for, foreach, break, continue
- Pattern matching:
  if (obj is string s) { ... }
  switch (shape) { case Circle c: ...; case Rectangle r when r.Width>r.Height: ...; }

3.4 Operators
- Arithmetic, logical, bitwise, null-coalescing (??), null-conditional (?.), null-coalescing assignment (??=), pattern matching, is, as, await, lambda =>

3.5 Example: small program

```
  using System;
  class Program {
    static void Main() {
      Console.WriteLine("Sum: " + Sum(3,4));
    }
    static int Sum(int a,int b) => a+b;
  }
```

4. BEGINNER
4.1 Object-Oriented Programming (OOP)
- Class vs struct:
  - Classes: reference semantics, allocated on managed heap (subject to GC), support inheritance
  - Structs: value semantics, shallow copy by value, stack allocation possibility, must be small and immutable for best use
- Inheritance:
```
  class Animal { public virtual void Speak() {} }
  class Dog : Animal { public override void Speak() => Console.WriteLine("Woof"); }
```
- Interfaces:
```
  interface IRepository<T> { T Get(int id); }
  class Repo : IRepository<User> { public User Get(int id) => ...; }
```
- Encapsulation: properties:
```
  public string Name { get; set; } // auto-property
  private int _age;
  public int Age { get => _age; private set => _age = value; }
```
- Records (C# 9+): succinct immutable types for DTOs:
```
  public record Person(string FirstName, string LastName);
```

4.2 Methods & Overloading
- Parameter modifiers: ref, out, in (readonly ref), params
- Optional/default parameters: void Log(string msg = "", int level = 1)
- Named arguments: Log(msg: "hi", level: 2)

4.3 Collections
- Arrays: T[] arr = new T[10];
- System.Collections.Generic: List<T>, Dictionary<TKey,TValue>, HashSet<T>, Queue<T>, Stack<T>
- Initialization:
```
  var list = new List<int> {1,2,3};
```
- Iteration: foreach (var item in list) { ... }
- When to choose: List<T> for resizable array; LinkedList<T> seldom needed; Dictionary for key-value.

4.4 Exceptions & Error Handling
- Throw and catch:
```
  try {
    // work
  } catch (ArgumentNullException ex) {
    // handle
  } finally {
    // cleanup
  }
```
- Create custom exceptions by deriving from Exception and implement serializable

constructors if public library.

4.5 Basic LINQ
- Using System.Linq; operations are deferred on IEnumerable<T> until enumerated; use ToList/ToArray to force evaluation.
- Examples:
  var big = people.Where(p => p.Age > 30).OrderBy(p => p.Name).Select(p => p.Email).ToList();

5. MEDIOCRE (LOWER-INTERMEDIATE)
5.1 Generics
- Class/interface/method generics:
  public class Box<T> { public T Value { get; set; } }
  public TOutput Map<TInput, TOutput>(TInput input) => ...;
- Constraints:
  where T : class
  where T : struct
  where T : new()
  where T : BaseType
- Avoid overusing reflection on generics; prefer compile-time constraints.

5.2 Delegates, Lambdas, and Events
- Delegates:
  public delegate int Transformer(int x);
  var t = new Transformer(x => x*2);
- Func and Action:
  Func<int,int> f = x => x + 1;
  Action<string> printer = s => Console.WriteLine(s);
- Events pattern:
  public event EventHandler<MyArgs> Changed;
  Changed?.Invoke(this, new MyArgs(...));
- Use weak event patterns or unsubscribe to avoid memory leaks.

5.3 Interfaces & Abstracts
- Prefer interfaces for dependencies; abstract classes when common code shared.
- Explicit interface implementation to avoid name collisions:
  void IFoo.Do() { ... }

5.4 Async/Await (basics)
- Use Task for async operations:
  async Task<int> GetAsync() { await Task.Delay(100); return 5; }
- Avoid async void except for top-level event handlers.
- Use ConfigureAwait(false) in libraries: await something.ConfigureAwait(false);

5.5 Common Pitfalls
- Boxing/unboxing (value types to object)
- Using exceptions for flow control
- Not disposing of IDisposable (use using or await using)
- Not understanding deferred execution in LINQ (closing over variables in loops)

6. NORMAL (INTERMEDIATE)
6.1 Advanced LINQ & Performance
- Use IEnumerable<T> vs IQueryable<T> appropriately.
- Prefer streaming (IEnumerable) for large sequences, but materialize when

reusing results.
- Avoid repeated enumerations; use ToList() if needed.

6.2 Async/Concurrency Patterns
- Task combinators: Task.WhenAll, Task.WhenAny
- Cancellation: use CancellationToken in async methods
- Use Concurrent collections for multi-threaded access: ConcurrentDictionary, ConcurrentQueue
- Synchronization: lock keyword, SemaphoreSlim for async locking, ReaderWriterLockSlim for read-heavy workloads

6.3 Dependency Injection (DI)
- Use Microsoft.Extensions.DependencyInjection:
  var services = new ServiceCollection();
  services.AddTransient<IMyService, MyService>();
  var sp = services.BuildServiceProvider();
- Lifetime choices: Transient, Scoped, Singleton (understand shared state)
- Use constructor injection; avoid service locator anti-pattern.

6.4 Unit Testing & Mocks
- Use xUnit for tests, Moq for mocking (or NSubstitute)
- Arrange-Act-Assert pattern
- Test single responsibility; keep tests deterministic
- Use coverage tools and test data builders

6.5 Logging & Configuration
- Microsoft.Extensions.Logging for DI-friendly logging
- Use structured logging (message templates)
- Microsoft.Extensions.Configuration to load JSON, env vars, command-line

6.6 Serialization
- System.Text.Json (built-in): JsonSerializer.Serialize/Deserialize
- Newtonsoft.Json (Json.NET) for advanced scenarios
- Beware of circular references and preserve/reference handling

6.7 Reflection & Expression Trees
- Reflection: System.Reflection to inspect types at runtime; expensive—cache results
- Expression trees for building code at runtime (LINQ providers, dynamic queries)

7. EXPERT
7.1 Memory & Span<T> / Memory<T>
- Span<T> and ReadOnlySpan<T> allow safe stack-allocated and sliceable memory views without allocation.
- Memory<T> is heap-based and can be awaited; Span<T> cannot be boxed or captured by async/iterator.
- Use stackalloc for small temporary buffers:
  Span<byte> buffer = stackalloc byte[256];
- Use System.Buffers.ArrayPool<T>.Shared.Rent() to reuse large arrays.

7.2 ref structs & ref returns
- ref struct restricts usage (can't be boxed, can't be captured by lambda/async).

- Example:
  ref int GetRef(int[] arr, int idx) => ref arr[idx];

7.3 Unsafe code & P/Invoke
- Use unsafe keyword for pointers and fixed blocks.
- Interop with native libs via DllImport or System.Runtime.InteropServices; prefer SafeHandles for resource safety.

7.4 JIT, IL, and Runtime
- Understand JIT compilation tiers (RyuJIT optimizations, tiered compilation), inlining, and how method size affects inlining.
- Use dotnet/runtime diagnostics and tools like dotnet-trace, PerfView.
- Inspect IL with tools: ILSpy, dotnet ildasm, dnSpy.

7.5 Garbage Collector (GC)
- Generational GC (Gen0, Gen1, Gen2), Large Object Heap (LOH) behavior, pinning causes fragmentation, use ArrayPool to reduce allocations.
- For low-latency apps, evaluate server/workstation GC settings and GCSettings.LatencyMode.

7.6 Source Generators & Roslyn
- Roslyn analyzers and source generators to produce code at compile-time; used for boilerplate elimination and compile-time validation.

7.7 AOT and Native Compilation
- .NET Native, Native AOT; size and startup tradeoffs; ensure compatibility of reflection-heavy code.

7.8 High-Performance Patterns
- Avoid unnecessary allocations, use struct enumerators, use ref locals and ref returns, prefer ValueTask where appropriate, but beware of misuse.
- Micro-benchmark with BenchmarkDotNet.

8. PERFORMANCE & MEMORY
8.1 General rules
- Measure first (profilers, BenchmarkDotNet)
- Hot path optimization only after profiling
- Reduce allocations and GC pressure
- Use caching and pooling when appropriate
- Use algorithms & data structures with right complexity

8.2 Common optimizations
- Use StringBuilder for heavy string concatenation in loops
- For value types, avoid boxing: use generic overloads
- Span<T> for slicing without allocations
- Use readonly members wherever possible to allow better JIT optimizations
- Prefer foreach with arrays/Lists; for LINQ on hot path, consider manual loops.

9. DEBUGGING & DIAGNOSTICS
9.1 Tools
- Visual Studio Debugger: breakpoints, conditional breakpoints, watch, immediate window, Edit & Continue
- VSCode: debugger for .NET
- dotnet trace, dotnet dump, dotnet-counters, PerfView, Windows Performance

Recorder, lldb on Linux/Mac
- Logging: structured logs and correlation ids + log levels

9.2 Strategies
- Reproduce deterministically
- Use assertions (Debug.Assert) for invariants
- Attach to process, inspect threads, view call stacks
- Analyze memory dumps for leaks and GC roots

10. SECURITY & BEST PRACTICES
10.1 Avoid common mistakes
- Never concatenate user input into SQL; use parameterized queries or ORM with parameterization
- Validate and sanitize all inputs; assume hostile input
- Use HTTPS/TLS (Kestrel with certificates)
- Secrets: do not commit to source control; use secret stores (Azure Key Vault, AWS Secret Manager) or dotnet user-secrets for dev
- Avoid insecure deserialization

10.2 Cryptography
- Prefer high-level APIs in System.Security.Cryptography
- Use modern algorithms: AES-GCM, RSA-OAEP or better, ECDSA for signatures where applicable
- Understand key management life-cycle and rotation

11. DESIGN PATTERNS & ARCHITECTURE
11.1 Patterns
- Singleton, Factory, Repository, Unit of Work, Strategy, Observer (events), Adapter, Decorator, Command, Mediator (MediatR)
- Prefer composition over inheritance in many cases

11.2 Architecture styles
- Layered (Presentation, Business, Data)
- Clean Architecture / Hexagonal / Onion (separate domain/core from infra)
- Microservices: bounded contexts, messaging, eventual consistency (use message brokers)
- Monorepo vs polyrepo considerations

11.3 SOLID & Clean Code
- Single Responsibility, Open/Closed, Liskov, Interface Segregation, Dependency Inversion
- Write testable, readable code; prefer descriptive names and small methods

12. BUILD, PACKAGING, CI/CD, NUGET
12.1 CI/CD tips
- Use dotnet build --configuration Release, dotnet test --no-build
- Use GitHub Actions, Azure DevOps pipelines, GitLab CI
- Cache NuGet packages and dotnet SDKs for speed

12.2 NuGet
- Create packages with dotnet pack or include <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
- Semantic versioning, symbols and source debugging (snupkg)
- Secure your package feed and use signed packages if required

12.3 Docker
- Use multi-stage builds to minimize final image size
- Choose base images: mcr.microsoft.com/dotnet/aspnet:8.0 for runtime, sdk images for build

13. COMMON APIS & LIBRARIES
- Web: ASP.NET Core (Kestrel, Middleware pipeline, Controllers, Minimal APIs)
- Data access: Entity Framework Core, Dapper for micro-ORM
- Messaging: RabbitMQ, Kafka clients, Azure Service Bus
- HTTP clients: HttpClient (use IHttpClientFactory)
- Serialization: System.Text.Json, Newtonsoft.Json
- DI: Microsoft.Extensions.DependencyInjection
- Logging: Serilog, NLog, Microsoft.Extensions.Logging

14. APPENDIX
14.1 Useful CLI Examples
- Create solution and projects:
  dotnet new sln -n SlnName
  dotnet new webapi -n Api
  dotnet sln add Api/Api.csproj
- Format code:
  dotnet format
- Run tests:
  dotnet test

14.2 Common Exceptions (and causes)
- NullReferenceException: dereferencing null
- ArgumentNullException: bad argument
- InvalidOperationException: method called in wrong object state
- TaskCanceledException: cancellation token triggered
- OutOfMemoryException: memory exhaustion — analyze allocations

14.3 C# Keyword Quick Reference (selected)
- Declaration: class, struct, interface, enum, record
- Access: public, private, protected, internal, protected internal, private protected
- Modifiers: static, readonly, const, volatile, sealed, abstract, virtual, override, async, unsafe, partial
- Flow: if, else, switch, case, default, for, foreach, while, do, break, continue, return, yield, throw
- Operators: new, this, base, is, as, sizeof, typeof, checked, unchecked, lock
- Nullable: ?, ??, ??=, ! (null-forgiving), #nullable

14.4 Idioms & Tips
- Prefer immutability where possible (records, readonly fields)
- Keep methods small and focused
- Favor constructor injection for dependencies
- Avoid redundant async/await (return Task directly when possible)
- Use pattern matching and expression-bodied members to improve readability
- Prefer exceptions for truly exceptional conditions

14.5 Example Patterns & Snippets
- Safe IDisposable usage:

```
  using (var resource = new Resource()) {
    // use
  }

- Async with cancellation:
  async Task<string> DownloadAsync(HttpClient client, CancellationToken ct) {
    using var resp = await client.GetAsync("url", ct).ConfigureAwait(false);
    resp.EnsureSuccessStatusCode();
    return await resp.Content.ReadAsStringAsync(ct).ConfigureAwait(false);
  }

- Implementing IDisposable (recommended pattern):
  public class MyResource : IDisposable {
    private bool _disposed;
    public void Dispose() => Dispose(true);
    protected virtual void Dispose(bool disposing) {
      if (_disposed) return;
      if (disposing) {
        // free managed
      }
      // free unmanaged
      _disposed = true;
    }
    ~MyResource() => Dispose(false);
  }

- Example using Span to parse bytes:
  int ParseInt(ReadOnlySpan<char> span) {
    int value = 0;
    foreach (char c in span) {
      value = value*10 + (c - '0');
    }
    return value;
  }

15. RESOURCES
- Microsoft docs: https://learn.microsoft.com/dotnet/csharp/
- .NET GitHub: https://github.com/dotnet
- Books:
  - C# in Depth (Jon Skeet)
  - CLR via C# (Jeffrey Richter)
  - Pro ASP.NET Core
- Tools & libs: BenchmarkDotNet, Serilog, AutoMapper, MediatR, Polly

16. CUSTOM CHECKLISTS (by level)
- Beginner checklist
  - Know project creation (dotnet new)
  - Understand classes, methods, arrays, lists
  - Use exceptions and try/catch
  - Write simple LINQ queries

- Mediocre checklist
  - Use generics, delegates, events
  - Implement interfaces and abstractions
```

  - Understand async/await and CancellationToken
  - Use basic DI

- Normal checklist
  - Write unit tests & use mocking
  - Profile and optimize hot paths
  - Deep understanding of EF Core or chosen persistence
  - CI/CD automation + logging and metrics

- Expert checklist
  - Use Span/Memory/ArrayPool effectively
  - Write Roslyn analyzers or source generators
  - Understand JIT and GC internals; know when to tune
  - Design scalable architectures and migrations
  - Secure, performant interop and native interactions

17. FINAL NOTES
- Practice: read source of .NET runtime and libraries, contribute to OSS.
- When optimizing, always measure.
- When building libraries, prefer simple, well-documented APIs, version carefully, and maintain backwards compatibility when possible.
- If you want, I can:
  - Expand any section into a full tutorial
  - Produce a printable cheat-sheet (.pdf) or smaller quick-reference .txt
  - Create sample projects demonstrating patterns (console, webapi, worker)
  - Generate unit-test templates or CI pipeline YAML

End of file.