C# Programming

Dr.-Ing. Grigory Devadze

September 5, 2025

* :BEAMER$_{env}$: frame :BEAMER$_{envargs}$: [plain]

Interfaces

Motivation

Why Interfaces?

- Define contracts between classes
- Separate what is done from how it is done
- Enable testability and interchangeability of components
- Foundation for Dependency Injection and loose coupling

Definition and Syntax

Interface Syntax in C#

```
public interface ILogger
{
    void Log(string message);
}
```

Implementation

```
public class ConsoleLogger : ILogger
{
    public void Log(string message) =>
        Console.WriteLine(message);
}
```

Demo

Example Program

```
public static class ProgramInterfaceDemo
{
    public static void Main()
    {
        ILogger log = new ConsoleLogger();
        log.Log("Hello from Interface demo!");
    }
}
```

Multiple Inheritance

Example

```
public interface IShape { double Area(); }
public interface IColored { string Color { get; } }

public class Circle : IShape, IColored
{
    public double Radius { get; set; }
    public string Color { get; set; } = "red";
    public double Area() =>
        Math.PI * Radius * Radius;
}
```

Practical Scenarios

Typical Use Cases

- Logging (ILogger $\rightarrow$ FileLogger, ConsoleLogger, RemoteLogger)
- Repositories (IDataRepository $\rightarrow$ SqlRepository, InMemoryRepository)
- UI/Events (IObserver / IObservable)
- Strategy pattern and plug-in systems

Naming Conventions

Why the “I” in ILogger?

- In C#, interface names are conventionally prefixed with I
    - Examples: ILogger, IDisposable, IEnumerable
- Origin: COM (Component Object Model) and early .NET guidelines
- Helps distinguish interfaces from classes at a glance
- Not enforced by the compiler → you could write “Logger”
- But: community and framework standards expect “I” prefix

Loose Coupling

What is Coupling?

- Coupling = how tightly two classes depend on each other
- Tight coupling: a class directly uses another concrete class
- Loose coupling: a class only depends on an interface, not a specific implementation

Example: Tight Coupling ()

```
public class ReportService
{
    private readonly ConsoleLogger _logger = new ConsoleLogger();

    public void Generate()
    {
        _logger.Log("Generating report...");
    }
}
```

Loose Coupling

Example: Loose Coupling ()

```
public class ReportService
{
    private readonly ILogger _logger;
    public ReportService(ILogger logger) => _logger = logger;

    public void Generate()
    {
        _logger.Log("Generating report...");
    }
}
```

Benefits

- Flexible: swap ConsoleLogger → FileLogger → RemoteLogger
- Easier to maintain and extend
- Foundation for Dependency Injection

Dependency Injection

What is Dependency Injection (DI)?

- Dependency = a required component (e.g., ILogger)
- Injection = passing the dependency from outside instead of creating it inside
- Promotes loose coupling and testability
- Widely used with DI frameworks (e.g., Microsoft.Extensions.DependencyInjection, Autofac)

Providers vs Consumers

Who depends on whom?

- Provider: implements the interface (e.g., ConsoleLogger : ILogger)
- Consumer: uses the interface (e.g., ReportService needs ILogger)
- Goal: Consumers depend on abstractions, not concrete classes

Example: Constructor Injection

```
public class ReportService
{
    private readonly ILogger _logger;

    // Dependency injected here
    public ReportService(ILogger logger)
    {
        _logger = logger;
    }

    public void Generate()
    {
        _logger.Log("Generating report...");
    }
}
```

Dependency Injection

```
// Usage
ILogger log = new ConsoleLogger();
var service = new ReportService(log);
service.Generate();
```

Dependency Injection

Benefits

- Easily swap implementations (ConsoleLogger $\rightarrow$ FileLogger $\rightarrow$ RemoteLogger)
- Enables unit testing with mock loggers
- Forms the foundation of modern .NET applications

Interfaces vs Abstract Classes

Conceptual Differences

- Interface = contract (what must be done)
- Abstract class = contract + shared implementation + state
- Inheritance:
  - Interfaces → multiple allowed
  - Abstract class → single base class only
- State/constructors:
  - Interfaces → no instance fields
  - Abstract classes → can have fields, constructors, protected helpers

Syntax Overview

```
public interface IWorker
{
    void DoWork();
    // since C# 8: default method implementations allowed
    void Report() => Console.WriteLine("Work done.");
}

public abstract class WorkerBase
{
    protected readonly string Name;
    protected WorkerBase(string name) => Name = name;

    public abstract void DoWork();        // must override
    public virtual void Report() =>       // may override
        Console.WriteLine($"{Name}: done.");
}
```

When to Choose Which?

- Choose interface when:
  - You need a contract only; implementations vary widely
  - You want consumers to be loosely coupled (DI)
  - Multiple, orthogonal capabilities are needed (e.g., `ILogger`, `IAsyncDisposable`)
- Choose abstract class when:
  - Implementations share state and non-trivial common code
  - You need protected helpers, constructors, or template methods
  - You want to evolve behavior in a single place

Advanced: Explicit Interface Implementation

Why?

- To avoid name clashes (two interfaces with same method name)
- To hide interface members from the public API

```
public interface IPrintable { void Print(); }
public class Report : IPrintable
{
    void IPrintable.Print() =>
        Console.WriteLine("Report printed");
}

var r = new Report();
// r.Print(); //  not visible
((IPrintable)r).Print(); //  works
```

Interfaces in .NET Ecosystem

Heavily used in .NET Core / ASP.NET Core

- `ILogger` → Logging abstraction
- `IConfiguration` → App settings
- `IServiceCollection` → Dependency Injection
- `IEnumerable<T>` → Collections and LINQ
- `IAsyncDisposable` → Async cleanup

Takeaway

- Most modern .NET features are built on interfaces
- Learning to design with interfaces = understanding the .NET ecosystem

Orthogonal Capabilities

Combining independent behaviors

```
public interface IPrintable { void Print(); }
public interface ISavable   { void Save(string path); }

public class Report : IPrintable, ISavable
{
    public void Print() => Console.WriteLine("Printing...");
    public void Save(string p) => File.WriteAllText(p, "data");
}
```

- Interfaces let you compose independent features without deep inheritance

Pitfalls

"I-itis"

- Defining interfaces just for the sake of it (`IDoSomething`)
- Abstractions without a clear purpose add complexity

Over-abstraction

- Don't create interfaces until you need multiple implementations
- Prefer YAGNI (You Aren't Gonna Need It)
- Abstract class is safer if API must evolve often

* :BEAMER$_{\text{env}}$: frame :BEAMER$_{\text{envargs}}$: [plain]

Events

Motivation

Why Events?

- Mechanism for publish/subscribe communication
- Enables loose coupling between components
- Core to many .NET frameworks (UI, I/O, async notifications)

Basics

```
public class Publisher
{
    public event EventHandler? SomethingHappened;

    public void DoWork()
    {
        Console.WriteLine("Working...");
        SomethingHappened?.Invoke(this, EventArgs.Empty);
    }
}

public class Subscriber
{
    public void OnSomething(object? sender, EventArgs e) =>
        Console.WriteLine("Received event!");
}
```

Demo

Example Program

```
public static class ProgramEventDemo
{
    public static void Main()
    {
        var pub = new Publisher();
        var sub = new Subscriber();

        pub.SomethingHappened += sub.OnSomething;

        pub.DoWork();  // Triggers event
    }
}
```

Custom EventArgs

Passing additional data

```
public class DataEventArgs : EventArgs
{
    public string Message { get; }
    public DataEventArgs(string msg) => Message = msg;
}
public class Publisher
{
    public event EventHandler<DataEventArgs>? DataAvailable;

    public void Publish(string msg) =>
        DataAvailable?.Invoke(this, new DataEventArgs(msg));
}
```

Built-in Example

FileSystemWatcher

```
using System.IO;

var watcher = new FileSystemWatcher(".");
watcher.Created += (s,e) => Console.WriteLine($"File created: {e.Name}"
watcher.EnableRaisingEvents = true;

Console.WriteLine("Press Enter...");
Console.ReadLine();
```

Built-in Example

Timer

```
using var timer = new System.Timers.Timer(1000); // 1s
timer.Elapsed += (s,e) => Console.WriteLine("Tick");
timer.Start();

Console.ReadLine();
```

Anonymous Event Handlers

```
public class ReportService
{
    private readonly ILogger _logger;

    // Dependency injected here
    public ReportService(ILogger logger)
    {
        _logger = logger;
    }

    public void Generate()
    {
        _logger.Log("Generating report...");
    }
}
```

Unsubscribing

Multiple Subscribers

Multicast Delegates

- An event can notify multiple subscribers
- Subscribers are invoked in order of subscription
- All receive the same event arguments

```
pub.SomethingHappened += (s,e) => Console.WriteLine("Handler 1");
pub.SomethingHappened += (s,e) => Console.WriteLine("Handler 2");
pub.DoWork();
```

Output

```
Working...
Handler 1
Handler 2
```

Takeaways

Events in C#

- Safe, built-in publish/subscribe pattern
- Provide loose coupling between components
- Can have multiple subscribers (multicast)
- Used throughout .NET (UI, I/O, diagnostics, async notifications)
- Always unsubscribe in long-running apps to avoid memory leaks

Asynchronous Programming

Motivation

Why Asynchronous Programming?

- Improve responsiveness (UI, servers stay reactive)
- Scale with I/O-bound work (network, disk, database)
- Avoid blocking threads unnecessarily
- Enable modern patterns: streaming or microservices

Sync vs Async vs Parallel

- Synchronous: one thing at a time, blocking
- Asynchronous: start an operation, continue without waiting
- Parallel: multiple threads doing work at the same time
- Async != Parallel!

Basics

Tasks

- `Task` represents an ongoing operation
- `Task<T>` represents an operation that returns a value
- Tasks can be awaited, combined, or cancelled

Async/Await Keywords

```
public async Task<int> ComputeAsync()
{
    await Task.Delay(1000); // simulate async work
    return 42;
}
```

Demo

```
public static class ProgramAsyncDemo
{
    public static async Task Main()
    {
        Console.WriteLine("Starting...");
        int result = await ComputeAsync();
        Console.WriteLine($"Result = {result}");
    }

    static async Task<int> ComputeAsync()
    {
        await Task.Delay(1000);
        return 0;
    }
}
```

Async I/O

File I/O

```
string text = await File.ReadAllTextAsync("data.txt");
Console.WriteLine($"Read {text.Length} chars");
```

HTTP Requests

```
using var http = new HttpClient();
string html = await http.GetStringAsync("https://example.com");
Console.WriteLine(html.Substring(0,100));
```

Combinators

WhenAll and WhenAny

```
var t1 = http.GetStringAsync("https://example.com");
var t2 = http.GetStringAsync("https://dotnet.microsoft.com");

await Task.WhenAll(t1, t2);   // wait for both
Console.WriteLine(t1.Result.Length + t2.Result.Length);

var first = await Task.WhenAny(t1, t2); // first to finish
Console.WriteLine("First completed!");
```

Error Handling

Exceptions in Async

```
try
{
    await DangerousAsync();
}
catch(Exception ex)
{
    Console.WriteLine($"Caught: {ex.Message}");
}
```

AggregateException

- Multiple exceptions can occur (e.g., `Task.WhenAll`)
- Flattened automatically when awaiting
- Example: `await Task.WhenAll(tasks)` throws the first exception

CancellationToken

```
var cts = new CancellationTokenSource();

var task = Task.Run(async () =>
{
    while (true)
    {
        cts.Token.ThrowIfCancellationRequested();
        await Task.Delay(500);
    }
}, cts.Token);

// cancel after 2s
cts.CancelAfter(2000);
try { await task; }
catch(OperationCanceledException) { Console.WriteLine("Cancelled!"); }
```

Timeouts and Throttling

Timeout

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(3));
await http.GetStringAsync("https://slow.example", cts.Token);
```

Throttling Parallelism

```
var urls = new [] { "a", "b", "c" };
using var sem = new SemaphoreSlim(3);
var tasks = urls.Select(async url =>
{
    await sem.WaitAsync();
    try
    {   var data = await http.GetStringAsync(url);
        Console.WriteLine($"{url}: {data.Length}");
    }
    finally { sem.Release(); }
});
await Task.WhenAll(tasks);
```

IProgress<T>

```
public async Task DownloadAsync(IProgress<int> progress)
{
    for(int i=0; i<=100; i+=10)
    {
        await Task.Delay(100);
        progress.Report(i);
    }
}

var p = new Progress<int>(v => Console.WriteLine($"{v}%"));
await DownloadAsync(p);
```

Async Streams

`IAsyncEnumerable<T>`

```
public async IAsyncEnumerable<int> NumbersAsync()
{
    for (int i=0; i<5; i++)
    {
        await Task.Delay(200);
        yield return i;
    }
}

await foreach(var n in NumbersAsync())
    Console.WriteLine(n);
```

Common Pitfalls

Pitfalls to Avoid

- Blocking async code with `.Result` or `.Wait()` $\rightarrow$ deadlocks!
- Forgetting `await` $\rightarrow$ tasks run but exceptions disappear
- Mixing sync + async badly $\rightarrow$ thread pool starvation
- Ignoring CancellationTokens

Best Practices

Guidelines

- Use `async/await` all the way down
- Prefer `ConfigureAwait(false)` in libraries
- Always support cancellation if possible
- Keep async methods non-blocking

Takeaways

Asynchronous Programming in C#

- Async/await simplifies non-blocking code
- Tasks model ongoing work, can be awaited, combined, cancelled
- Great for I/O-bound scalability
- Not the same as threading
- Foundation for reactive and streaming systems

Threading

Motivation

Why Threading?

- Exploit multiple CPU cores for concurrency
- Run multiple operations simultaneously
- Keep applications responsive (background work)
- Essential for parallel computing and servers

Async vs Threading

- Asynchronous: efficient I/O, non-blocking
- Threading: true parallelism for CPU-bound work
- Often combined in modern apps

Basics

Creating Threads

```
using System.Threading;

Thread t = new Thread(() =>
{
    Console.WriteLine("Hello from worker thread!");
});
t.Start();
t.Join(); // wait for completion
```

Output

```
Hello from worker thread!
```

ThreadPool

ThreadPool and Tasks

- Creating raw threads is expensive
- .NET provides a ThreadPool for short-lived tasks
- Usually accessed via `Task.Run`

```
Task t = Task.Run(() =>
{
    Console.WriteLine("Running on ThreadPool");
});
t.Wait();
```

Race Conditions

```
int counter = 0;

Parallel.For(0, 10000, i =>
{
    counter++; // not thread-safe!
});
Console.WriteLine(counter); // not 10000
```

Why? Multiple threads update the same memory concurrently

Lock

```
object locker = new();
int counter = 0;

Parallel.For(0, 10000, i =>
{
    lock(locker)
    {
        counter++;
    }
});
Console.WriteLine(counter); // always 10000
```

Interlocked

```csharp
int counter = 0;

Parallel.For(0, 10000, i =>
{
    Interlocked.Increment(ref counter);
});
Console.WriteLine(counter); // always 10000
```

Benchmarks

Lock vs Interlocked vs No Sync

- Example results (N=5,000,000, 12 cores)
  - lock: a=5000000, time=156 ms
  - Interlocked: b=5000000, time=124 ms
  - NO sync (!!): c=417251, time=38 ms

Insights

- Correctness > performance
- Interlocked is faster for simple counters
- Locks allow more complex critical sections

ReaderWriterLockSlim

Many readers, few writers

```
var dict = new Dictionary<int,int>();
var rw = new ReaderWriterLockSlim();

Parallel.For(0, 10000, i =>
{
    if (i % 10 == 0)
    {
        rw.EnterWriteLock();
        try { dict[i] = i; }
        finally { rw.ExitWriteLock(); }
    }else
    {
        rw.EnterReadLock();
        try { _ = dict.ContainsKey(i - 1); }
        finally { rw.ExitReadLock(); }
    }
});
```

Thread-Local Storage

ThreadLocal<T>

```
var local = new ThreadLocal<int>(() => 0);

Parallel.For(0, 1000, i =>
{
    local.Value++;
    Console.WriteLine($"Thread \
                      {Thread.CurrentThread.ManagedThreadId}: \
                      {local.Value}");
});
```

Each thread maintains its own independent value

Pitfalls

Common Issues

- Race conditions (unprotected shared state)
- Deadlocks (threads waiting forever)
- Starvation (some threads never scheduled)
- Excessive threads $\rightarrow$ overhead, context switching

Best Practices

Guidelines

- Prefer `Task.Run` and the ThreadPool for most work
- Use `lock` for critical sections, `Interlocked` for counters
- Use `ReaderWriterLockSlim` for read-heavy scenarios
- Avoid creating too many raw threads
- Combine with async for modern scalable systems

Takeaways

Threading in C#

- Enables true CPU parallelism
- Threads are heavy, prefer the ThreadPool
- Synchronization is essential for correctness
- Always balance correctness vs performance
- Foundation for higher-level abstractions (TPL, Parallel LINQ, async/await)

Diagnostics

Motivation

Why Diagnostics?

- Production systems need visibility:
  - Debugging errors
  - Performance bottlenecks
  - Monitoring live behavior
- `Console.WriteLine` is not enough
- Structured logging and events enable tools and automation

Key Concepts

- Logging = what happened
- Tracing = sequence of events
- Metrics = numeric measurements over time

Tracing Basics

System.Diagnostics

```
using System.Diagnostics;

Trace.Listeners.Add(new TextWriterTraceListener("log.txt"));

Trace.WriteLine("Application started");
Trace.TraceWarning("Low memory");
Trace.TraceError("Something went wrong");
Trace.Flush();
```

Notes

- Works with multiple listeners (console, file, custom)
- Simple, but limited compared to modern logging

Structured Events (ETW-style)

```csharp
using System.Diagnostics.Tracing;

[EventSource(Name = "MyApp-Events")]
public sealed class MyEventSource : EventSource
{
    public static readonly MyEventSource Log = new();

    [Event(1, Level=EventLevel.Informational)]
    public void RequestStart(string route) => WriteEvent(1, route);

    [Event(2, Level=EventLevel.Informational)]
    public void RequestStop(int status) => WriteEvent(2, status);

    [Event(3, Level=EventLevel.Error)]
    public void Failure(string message) => WriteEvent(3, message);
}
```

EventSource

Usage

```
MyEventSource.Log.RequestStart("/api");
MyEventSource.Log.Failure("DB connection failed");
```

Tools

- Capture with `dotnet-trace collect --providers MyApp-Events`
- View in PerfView or Visual Studio diagnostics

Logging Abstractions

ILogger in .NET Core

```
using Microsoft.Extensions.Logging;

var factory = LoggerFactory.Create(b => b.AddConsole());
ILogger log = factory.CreateLogger("Demo");

log.LogInformation("Started processing");
log.LogWarning("Something looks odd");
log.LogError("An error occurred");
```

Benefits

- Unified abstraction
- Multiple providers (console, file, cloud, Application Insights)
- Configurable log levels

EventCounter API

```
public sealed class MetricsSource : EventSource
{
    public static readonly MetricsSource Log = new("MetricsDemo");

    private EventCounter _reqPerSec;

    public MetricsSource(string name) : base(name)
    {
        _reqPerSec = new EventCounter("requests-per-second", this);
    }

    public void Report(double value) => _reqPerSec.WriteMetric(value);
}
```

Metrics and Counters

Tools

- `dotnet-counters monitor MetricsDemo`
- Real-time metrics: request rate, latency, memory usage

DiagnosticSource

Low-level Instrumentation

```
using System.Diagnostics;

var source = new DiagnosticListener("MyLibrary");

if (source.IsEnabled("RequestStart"))
    source.Write("RequestStart", new { Url = "/api", Method = "GET" });
```

Observers

- Libraries emit diagnostic events
- Observability systems (OpenTelemetry, Application Insights) subscribe
- Decouples producers and consumers

Practical Tools

.NET CLI Tools

- `dotnet-trace` → collect EventSource + runtime events
- `dotnet-counters` → monitor performance counters live
- `dotnet-dump` → capture and analyze process memory dumps
- PerfView → powerful UI for ETW / EventPipe traces

Example

```
dotnet-trace collect --process-id 1234 --providers MyApp-Events
dotnet-counters monitor System.Runtime
```

Takeaways

Diagnostics and Instrumentation

- Logging, tracing, metrics = 3 pillars of observability
- Use `ILogger` for app-level logging
- Use `EventSource` and `EventCounter` for structured diagnostics
- Use `DiagnosticSource` for library instrumentation
- .NET tools (trace, counters, dump, PerfView) provide visibility into live systems

Reflection

Motivation

Why Reflection?

- Inspect types, methods, properties at runtime
- Dynamically invoke members
- Discover metadata (attributes, interfaces)

Why Attributes?

- Declaratively attach metadata to code
- Frameworks use attributes to drive behavior
- Cleaner than hardcoding configuration in code

Reflection Basics

Get Type Information

```
Type t1 = typeof(string);
Type t2 = "hello".GetType();

Console.WriteLine(t1.FullName); // System.String
Console.WriteLine(t2.IsClass);  // True
```

Inspect Members

```
Type t = typeof(DateTime);

foreach(var m in t.GetMethods())
    Console.WriteLine(m.Name);
```

Reflection: Dynamic Invocation

Invoke a Method

```
Type t = typeof(Math);
object? result = t.InvokeMember("Sqrt",
    BindingFlags.InvokeMethod
    | BindingFlags.Public
    | BindingFlags.Static,
    binder: null,
    target: null,
    args: new object[] { 9.0 });

Console.WriteLine(result); // 3
```

Create Instance Dynamically

```
Type t = typeof(StringBuilder);
object obj = Activator.CreateInstance(t)!;
```

Assemblies

Load Types from Assembly

```
var asm = Assembly.GetExecutingAssembly();
foreach (var type in asm.GetTypes())
    Console.WriteLine(type.FullName);
```

Use Cases

- Plugin systems (discover types at runtime)
- Reflection-based DI containers
- Unit test discovery

Common Examples

```
[Obsolete("Use NewMethod instead")]
public void OldMethod() {}

[Serializable]
public class Person { }

public class Customer
{
    [Required]
    public string Name { get; set; }
}
```

Custom Attributes

Define Your Own

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class AuthorAttribute : Attribute
{
    public string Name { get; }
    public AuthorAttribute(string name) => Name = name;
}
```

Apply It

```
[Author("Me")]
public class Report { }
```

Reading Attributes

With Reflection

```
Type t = typeof(Report);
var attrs = t.GetCustomAttributes(typeof(AuthorAttribute),
                                  inherit: false);

foreach(AuthorAttribute a in attrs)
    Console.WriteLine(a.Name); // Me
```

Real-World Uses

Attributes in Frameworks

- ASP.NET Core routing:
  - `[HttpGet("/api/data")]`
- Entity Framework:
  - `[Key]`, `[Table("Users")]`
- NUnit/xUnit testing:
  - `[Test]`, `[Fact]`
- Serialization:
  - `[JsonProperty("id")]`

Benefits

- Declarative, less boilerplate
- Frameworks discover behavior via reflection
- Extensible $\rightarrow$ you can define your own

Pitfalls

Reflection

- Slower than direct code (metadata lookup)
- Can bypass encapsulation (dangerous if misused)
- Complicates code analysis and refactoring

Attributes

- Overuse can clutter code
- Hard to validate at compile time
- Framework coupling (depends on specific attributes)

Takeaways

Reflection and Attributes

- Reflection = runtime type inspection & dynamic invocation
- Attributes = declarative metadata
- Together they power many frameworks (ORMs, DI, test frameworks, serializers)
- Use reflection with care: correctness first, performance second
- Attributes are a clean way to configure behavior declaratively

Native Interop

Motivation

Why Native Interop?

- Reuse existing C/C++ libraries
- Access operating system APIs directly (Win32, libc)
- Performance: use optimized native code (e.g., BLAS, OpenSSL)
- Device integration (USB, PCIe, hardware APIs)

Approaches

- P/Invoke (Platform Invocation Services)
- COM Interop (legacy Windows components)
- C++/CLI bridge (mixed-mode assemblies)
- Focus: P/Invoke = most common & practical

P/Invoke Basics

Importing Native Functions

```
using System.Runtime.InteropServices;

public static class NativeMethods
{
    [DllImport("user32.dll", CharSet = CharSet.Unicode)]
    public static extern int MessageBox(
        IntPtr hWnd,
        string text,
        string caption,
        uint type);
}
```

Usage

```
NativeMethods.MessageBox(IntPtr.Zero,
                         "Hello from native code!",
                         "Interop Demo", 0);
```

Data Marshalling

Automatic Type Conversion

- P/Invoke marshals managed $\leftrightarrow$ unmanaged types
- Common cases:
  - `string` $\rightarrow$ LPWSTR / char*
  - `bool` $\rightarrow$ BOOL
  - `int, double` $\rightarrow$ `int, double`
- Attributes customize marshaling:
  - `[MarshalAs(UnmanagedType.LPStr)] string s;`

Structs

```
dotnet-trace collect --process-id 1234 --providers MyApp-Events
dotnet-counters monitor System.Runtime
```

Calling Native Code with P/Invoke

```
// Windows API (C)
DWORD GetTickCount(void);

// C# P/Invoke declaration
using System.Runtime.InteropServices;

public static class NativeMethods
{
    [DllImport("kernel32.dll")]
    public static extern uint GetTickCount();
}

Console.WriteLine(
    $"Ticks: {NativeMethods.GetTickCount()}"
);
```

Calling Native Code with P/Invoke

Notes

- ‘DWORD‘ = 32-bit unsigned integer (‘uint‘ in C#).
- Function returns system uptime in ms (wraps after ~49.7 days).
- Shows how WinAPI types map to C# types in interop.

Passing Delegates to Native Code

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
public delegate int Callback(int x);

[DllImport("NativeLib.dll")]
public static extern void RegisterCallback(Callback cb);

// Usage
RegisterCallback(x => {
    Console.WriteLine($"Callback {x}");
        return x * 2;
});
```

Marshaling delegates allows native → managed calls

Advanced Interop

Unsafe Code

```
unsafe
{
    int value = 42;
    int* ptr = &value;
    Console.WriteLine(*ptr); // dereference
}
```

Notes

- Needed for high-performance pointer manipulation
- Use sparingly in safe code bases

Real-World Scenarios

Where P/Invoke is Used

- Call Windows APIs (MessageBox, file operations)
- Bind to C libraries (e.g., SQLite, OpenSSL, LAPACK)
- Interact with hardware drivers
- Performance hotspots (SIMD, image processing)

Pitfalls

Interop Risks

- Crashes (bad pointers, invalid structs)
- Memory leaks (unmanaged allocations not GC'd)
- Performance overhead from marshaling
- Platform differences (x86/x64, Windows/Linux)

Best Practices

Guidelines

- Prefer managed APIs when available
- Isolate interop in dedicated classes (NativeMethods)
- Always match signatures exactly (calling convention, struct layout)
- Test on all target platforms
- Use safe handles for unmanaged resources

Takeaways

Native Interop in C#

- P/Invoke allows direct calls to native DLLs
- Marshaling bridges managed and unmanaged worlds
- Useful for OS APIs, performance, device integration
- Powerful but dangerous: handle with care
- Foundation for system programming with .NET

Garbage collection

Motivation

Why care about memory & GC?

- Throughput & latency depend on allocation patterns
- Latency-sensitive workloads (UI, trading, games) need predictable pauses
- Cloud efficiency: fewer GB $\rightarrow$ fewer €€
- Debuggability: find leaks, finalizer issues

Foundations

Value vs Reference types

- Value types (`struct`) usually live on the stack
- Reference types (class) live on the managed heap
- Boxing allocates: `object o = 1;` → avoid in hot paths

Stack vs Heap

- Stack: fast, scoped, no GC
- Heap: managed by GC, supports long-lived objects

.NET GC model

Generational GC

- Gen0 → short-lived objects
- Survivors → Gen1 → Gen2 (long-lived)
- Background GC for Gen2; ephemeral segments for Gen0/Gen1

Heaps

- SOH (Small Object Heap): < ~85 KB
- LOH (Large Object Heap): >= ~85 KB (collected with Gen2, prone to fragmentation)
- POH (Pinned Object Heap): isolates pinned blocks to reduce SOH/LOH fragmentation

Allocation patterns

Fast path & pitfalls

```
// Fast: small arrays/objects on SOH
var buf = new byte[1024];

// LOH allocation (>= ~85 KB) - more expensive & fragmented
var big = new byte[200_000];
```

- Many small, short-lived allocs → OK (Gen0 friendly)
- Many large allocs → LOH churn, consider pooling (reusing instead `new T[n]`)

Performance Pitfalls & Mitigations

LOH churn

- Frequent alloc/free cycle = "LOH churn" → more Gen2 collections, fragmentation

Mitigation: pooling

- Reuse large buffers instead of allocating each time
- `ArrayPool<T>.Shared.Rent(size)` → get array (may be larger than requested)
- Use the array, then `ArrayPool<T>.Shared.Return(array)`
- Optionally clear on return (`Return(array, clearArray:true)`) for security

Example

```
using System.Buffers;

// Rent a 100k-element array from the shared pool
int[] buffer = ArrayPool<int>.Shared.Rent(100_000);

try {
    buffer[0] = 1;
}
finally {
    // IMPORTANT: return for reuse
    ArrayPool<int>.Shared.Return(buffer);
}
```

Guidelines

- Avoid holding rented arrays for long periods
- Great for networking, serialization, image/audio/video processing
- Cuts down LOH allocations $\rightarrow$ fewer pauses, less fragmentation

Observing the GC

Collection counts

```
for (int gen = 0; gen <= GC.MaxGeneration; gen++)
    Console.WriteLine($"Gen{gen}: {GC.CollectionCount(gen)}");
```

Total managed memory

```
Console.WriteLine($"Managed: \
                  {GC.GetTotalMemory(false)/1024.0:N1} KB");
```

Forcing/controlling GC

For secrets (passwords, keys, tokens)

- Clear manually before releasing:
  - `Array.Clear(myArray)`
  - `CryptographicOperations.ZeroMemory(span)` (System.Security.Cryptography)
- Prefer `SecureString` or `Span<byte>` buffers you control

Latency modes & NoGCRegion

```
GCSettings.LatencyMode = GCLatencyMode.SustainedLowLatency;
// Critical section...
GCSettings.LatencyMode = GCLatencyMode.Interactive;

// Try NoGCRegion for short  time-critical windows
if (GC.TryStartNoGCRegion(10_000_000))
{ /* critical */ GC.EndNoGCRegion(); }
```

Server vs Workstation GC

Modes

- Workstation GC: desktop apps, responsive UI
- Server GC: multi-core servers, multiple heaps, higher throughput
- Configure in runtimeconfig.json or hosting (ASP.NET Core defaults to Server GC)

Finalization & Dispose

Finalizers are non-deterministic

```
class Demo : IDisposable {
    ~Demo() { /* non-deterministic, on finalizer thread */ }
    public void Dispose() { /* deterministic */ GC.SuppressFinalize(thi
}
```

- Prefer `IDisposable` / `using` (or `await using`) over finalizers
- Use `SafeHandle` instead of raw IntPtr for native resources

Spans & zero-alloc APIs

Span<T> / Memory<T> (stackalloc, ref struct)

```
Span<int> stackData = stackalloc int[128];
stackData[0] = 123;

// slicing without allocations
ReadOnlySpan<char> slice = "abcdef".AsSpan(1,3); // "bcd"
```

- Spans avoid heap allocations for slicing/parsing
- Great for serializers, parsing, IO pipelines

Strings & builders

Reduce string churn

```
var sb = new System.Text.StringBuilder();
for (int i=0; i<1000; i++) sb.Append(i).Append(',');
string s = sb.ToString(); // allocate once
```

- Avoid "+" in loops; prefer `StringBuilder`
- Consider `string.Create` for formatted output in hot paths

Async & allocations

State machines & ValueTask

```
public async ValueTask<int> FooAsync() { await Task.Yield(); return 1; }
```

- Each async method creates a state machine object when it awaits
- For frequently completing synchronously, consider `ValueTask`

Pinning & interop

Native interop needs stable pointers

Pinning hurts the GC

```
// Pinned memory can fragment the heap
var handle = GCHandle.Alloc(buffer, GCHandleType.Pinned);
// ...
handle.Free();
```

- Prefer `fixed Span<byte>` patterns or POH; minimize pin duration

Diagnostics toolbox

.NET CLI & tools

- `dotnet-counters monitor System.Runtime` (alloc rate, GC heap size, %time in GC)
- `dotnet-trace collect` (GC pauses, allocations)
- `dotnet-gcdump collect` & analyze in PerfView/VS
- Visual Studio: Diagnostic Tools (Memory Usage, Heap snapshots)
- PerfView: GC stats, LOH fragmentation, allocation stacks

Quick live check

```
dotnet-counters monitor --process-id <PID> System.Runtime
# GC Heap Size, Gen0/1/2 collections, Allocation Rate, % Time in GC
```

BenchmarkDotNet (recommended)

```
[MemoryDiagnoser]
public class AllocBench {
  [Benchmark] public int Boxing() { object o = 42; return (int)o; }
  [Benchmark] public int NoBox() { int x = 42; return x; }
}
```

- Reports time + allocations (Gen0/LOH) per operation

Common pitfalls

Watch out for

- Loops with hidden allocations (boxing, LINQ closures, foreach on non-struct enumerators)
- Large temporary arrays $\rightarrow$ LOH churn
- Long-lived event subscriptions $\rightarrow$ leaks (publisher holds subscriber)
- Finalizer backlog $\rightarrow$ high GC pause times
- Excessive small string concatenations

Practical patterns

Quick wins

- Reuse buffers (ArrayPool)
- Prefer structs for tiny immutable data on hot paths (avoid boxing)
- Avoid capturing lambdas in tight loops
- Consider `TryParse` over `Parse` (no exceptions → no allocations)
- Use logging templates (structured) instead of string interpolation in hot paths

Quick demos (copy-paste)

Gen counts & allocs

```
for (int i=0; i<10_000; i++) _ = new byte[1024];
Console.WriteLine($"Gen0:{GC.CollectionCount(0)} Gen1:{GC.CollectionCou
```

LOH impact

```
var sw = System.Diagnostics.Stopwatch.StartNew();
for (int i=0; i<200; i++) _ = new byte[100_000]; // LOH
sw.Stop();
Console.WriteLine($"Time: {sw.ElapsedMilliseconds} ms");
```

Configuration notes

GC settings

- Server vs Workstation GC (runtimeconfig.json, hosting)
- Heap hard limits in containers (respect cgroup memory)
- Environment variables: `COMPlus_GCHeapHardLimit`, `DOTNET_GCHeapHardLimit`

Takeaways

Memory & GC in .NET

- Generational GC favors many short-lived objects
- LOH/POH require special care (pooling, pinning)
- Prefer `IDisposable`, pooled buffers, spans, and zero-alloc APIs
- Measure with `BenchmarkDotNet`; observe with `dotnet-counters` / PerfView / VS
- Tune only after measuring; correctness & clarity first

Motivation

Why Serial Today?

- Industrial devices (PLCs, sensors, motor controllers)
- Embedded (MCUs, Arduino), lab instruments (SCPI)
- Simple, reliable, widely available (USB-to-Serial)

Quick Start

List Available Ports

```
using System;
using System.IO.Ports;

Console.WriteLine("Ports: " +
                  string.Join(", ", SerialPort.GetPortNames()));
```

Port Naming

- Windows: `COM1`, `COM3`, ...
- Linux: `/dev/ttyS0`, `/dev/ttyUSB0`, `/dev/ttyACM0`
- macOS: `/dev/tty.usbserial-xxx`, `/dev/tty.usbmodem-xxx`

Basic Open/Write/Read

Minimal Console Demo

```
using System.IO.Ports;

var portName = "COM3"; // or "/dev/ttyUSB0"
using var sp = new SerialPort(portName, 115200, Parity.None,
                              8, StopBits.One) {
      Handshake = Handshake.None,
      ReadTimeout = 1000,
      WriteTimeout = 1000,
      NewLine = "\r\n" // match your device!
    };
sp.Open();
Console.WriteLine($"Opened {sp.PortName} @ {sp.BaudRate} baud");
sp.WriteLine("IDN?");
var reply = sp.ReadLine();// blocks until NewLine or timeout
Console.WriteLine($"Reply: {reply}");
```

DataReceived Event

```
using System;
using System.IO.Ports;

public class Reader {
  private readonly SerialPort _sp;

  public Reader(string name) {
    _sp = new SerialPort(name, 115200) { NewLine = "\n" };
    _sp.DataReceived += OnData;
  }
```

Event-Driven Reading

```
  public void Start() => _sp.Open();
  public void Stop() { _sp.DataReceived -= OnData; _sp.Close(); }

  private void OnData(object? s,
                      SerialDataReceivedEventArgs e) {
    try {
      var line = _sp.ReadExisting(); // or ReadLine() if framed
      Console.Write(line);           // avoid long work in event
    } catch (TimeoutException) { }
  }
}
```

Notes

- Event can fire for partial data; buffer and parse accordingly
- Keep handler fast; offload to a queue/Task if needed

Async I/O (BaseStream)

Modern Pattern (no events)

```
public static class SerialAsync {
  public static async Task RunAsync(string port,
                                    CancellationToken ct) {
    using var sp = new SerialPort(port, 115200);
    sp.Open();
    var buf = new byte[4096];
    while (!ct.IsCancellationRequested) {
      int n = await sp.BaseStream
                  .ReadAsync(buf.AsMemory(0, buf.Length), ct);
      if (n == 0) break; // closed
      Console.Write(Encoding.ASCII.GetString(buf, 0, n));
    }
  }
}
```

Async I/O (BaseStream)

When to use

- High throughput
- Structured backpressure and cancellation
- Cleaner than events for streaming

Framing & Encoding

NewLine and Text Encoding

- Set `SerialPort.NewLine` to your device's terminator (`"\n"`, `"\r\n"`, `"\r"`)
- Default encoding is ASCII; change if needed:
  - `sp.Encoding = Encoding.UTF8;`
- For binary protocols: use `Read/Write(byte[], ...)` and avoid `ReadLine()`

Hex Utilities (binary debug)

```
static string Hex(byte[] a, int n) =>
  BitConverter.ToString(a, 0, n).Replace("-", " ");
```

Flow Control & Signals

Handshake / RTS / DTR

```
sp.Handshake = Handshake.None;
// or RequestToSend, XOnXOff, RequestToSendXOnXOff
sp.RtsEnable = true;
// manual RTS if needed
sp.DtrEnable = true;
// some devices require DTR high
```

Common Settings

- Baud: 9600, 19200, 38400, 57600, 115200
- Data bits: usually 8
- Parity: None/Even/Odd
- Stop bits: One/Two

Timeouts & Reliability

Timeouts

```
sp.ReadTimeout  = 1000; // ms, throws TimeoutException
sp.WriteTimeout = 1000;
```

- For robust protocols, implement retries and application-level checksums

Cross-Platform Notes

Windows vs Linux/macOS

- Windows: install USB-serial driver if needed; COM port number via Device Manager
- Linux: permissions — add user to `dialout` (or `uucp`) group, or `sudo`
- macOS: use `/dev/tty.usb*=/=cu.usb*`; sometimes need vendor drivers

Test Without Hardware

- Windows: com0com (virtual null-modem pair), Virtual Serial Driver Pro
- Linux/macOS: `socat -d -d pty,raw,echo=0 pty,raw,echo=0`

Threading & Safety

Access from One Thread

- `SerialPort` is not fully thread-safe for concurrent read/write/events
- Prefer a single I/O loop + channels/queues to communicate with UI/logic

Don't Block Event Thread

- In `DataReceived`, quickly hand off data

Proper Shutdown

```
sp.DataReceived -= OnData;
if (sp.IsOpen) sp.Close();
sp.Dispose();
```

- Close before Dispose; consider `using` where possible

Troubleshooting

Common Issues

- Wrong `NewLine` → `ReadLine()` hangs
- Wrong parity/stop bits/baud → garbage characters
- Flow control mismatch → stalled writes
- Linux permissions → "Access denied": add user to `dialout`, re-login
- USB sleep/low-power can drop the port on laptops

Takeaways

System.IO.Ports

- Simple API for serial comms, event-driven or async streaming
- Mind framing (text vs binary), flow control, and timeouts
- Prefer single-threaded I/O with clean handoff to app logic
- Test with virtual ports when hardware is unavailable

WDK Tutorial

Motivation

- Windows Driver Kit (WDK) = toolkit for building drivers
- Enables integration with hardware and creation of virtual devices
- Typical use cases:
  - Device drivers (USB, PCIe, Serial, Network)
  - Virtual drivers for testing
  - Filter drivers (extend behavior of existing devices)

Driver Types

- KMDF (Kernel-Mode Driver Framework)
  - Easier and safer than raw WDM
  - Standard choice for most device drivers
- UMDF (User-Mode Driver Framework)
  - Runs in user mode (safer, less powerful)
  - Good for sensors, simple USB devices
- Specialized: File system drivers, filter drivers

Workflow

- Develop driver code in Visual Studio (C, KMDF templates)
- Define device descriptors and logic (EvtDeviceAdd, queues, I/O)
- Provide an INF file → tells Windows how to install the driver
- Build → sign → install (test mode or WHQL for production)
- Debug with WinDbg/DebugView, trace with ETW

Minimal Driver (Hello WDK)

```c
#include <ntddk.h>
#include <wdf.h>

DRIVER_INITIALIZE DriverEntry;

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
                     PUNICODE_STRING RegistryPath)
{
    KdPrint(("Hello WDK! Driver loaded.\n"));
    return STATUS_SUCCESS;
}
```

- Logs appear in kernel debugger

INF Skeleton

```
[Version]
Signature="$WINDOWS NT$"
Class=Sample
ClassGuid={78A1C341-4539-11d3-B88D-00C04FAD5171}
Provider=%Mfg%
DriverVer=09/04/2025,1.0.0.0

[Manufacturer]
%Mfg%=Demo,NTamd64
```

INF Skeleton

```
[Demo.NTamd64]
%DeviceName%=Install, Root\HelloDriver

[Install]
CopyFiles=DriverCopy

[DriverCopy]
HelloDriver.sys

[Strings]
Mfg="Demo Drivers"
DeviceName="Hello WDK Driver"
```

Best Practices

- Start with KMDF templates
- Keep drivers small, move heavy logic to user mode
- Prefer to use test VMs
- Always include INF + signature (test-signed ok for dev)

Takeaways

- WDK = essential toolset for Windows drivers
- KMDF simplifies driver development
- Every driver has:
  - Code (.sys)
  - INF (install metadata)
  - Signature
- Tools: WinDbg, DebugView, TraceView
- Foundation for real hardware and virtual device projects

Motivation

- LINQ: Query syntax for objects, composable operators
- Parallelism: PLINQ (parallel LINQ) and TPL (Task Parallel Library)
- Goal: Write declarative, efficient and scalable C# code

LINQ Basics

- Query over `IEnumerable<T>` / `IQueryable<T>`
- Deferred execution until `ToList()` / iteration
- Pure functions preferred

```
var numbers = Enumerable.Range(1, 10);
var evensSquared = numbers
    .Where(n => n % 2 == 0)
    .Select(n => n * n);
```

LINQ Common Operators

- Filter / Map: `Where`, `Select`
- Order: `OrderBy`, `ThenBy`
- Aggregation: `Count`, `Sum`, `Average`
- Grouping: `GroupBy`, `ToLookup`
- Joins: `Join`, `GroupJoin`
- Quantifiers: `Any`, `All`

Group & Aggregate Example

```
var revenueByRegion = sales
    .GroupBy(s => s.Region)
    .Select(g => new {
        Region = g.Key,
        Revenue = g.Sum(s => s.Qty * s.Price)
    })
    .OrderByDescending(x => x.Revenue);
```

Joins

- Inner join with `Join`
- Left join with `GroupJoin` + `DefaultIfEmpty`

```
var left =
    from c in customers
    join o in orders on c.Id equals o.CustId into grp
    from o in grp.DefaultIfEmpty()
    select new { c.Name, Total = o?.Total ?? 0m };
```

PLINQ: Parallel LINQ

- `AsParallel()` starts parallelization
- Good for CPU-bound, independent data
- Control order: `AsOrdered()` vs `AsUnordered()`
- Merge options: buffered vs unbuffered

```
var primesPar = data
    .AsParallel()
    .WithDegreeOfParallelism(Environment.ProcessorCount)
    .Where(IsPrime)
    .Count();
```

PLINQ Features

- `ForAll` pushes results in parallel
- `WithCancellation(cts.Token)` supports cancellation
- Exceptions: wrapped in `AggregateException`

```
data.AsParallel()
    .Where(IsPrime)
    .ForAll(p => bag.Add(p));
```

TPL: Task Parallel Library

- `Parallel.For` / `Parallel.ForEach` for loops
- Thread-local accumulators
- Concurrency control: `ParallelOptions`

```
Parallel.For(0, arr.Length, i => arr[i] = i * i);

var opts = new ParallelOptions { MaxDegreeOfParallelism = 4 };
Parallel.ForEach(files, opts, file => Process(file));
```

Tasks & Async

- Use `Task` + `async/await` for I/O concurrency
- Not suited for PLINQ

```
var downloads = urls.Select(async u => await http.GetStringAsync(u));
string[] pages = await Task.WhenAll(downloads);
```

When to Use What

- LINQ → Declarative, sequential queries
- PLINQ → CPU-heavy, large, independent data
- TPL → Custom loops, fine-grained control
- Task/async → I/O-bound workloads

Rules of Thumb

- Small data $\rightarrow$ sequential LINQ
- Heavy per-item CPU $\rightarrow$ PLINQ
- I/O $\rightarrow$ async/await Tasks
- Avoid shared state, prefer pure functions
- Measure performance!

Summary

- LINQ simplifies data queries in C#
- PLINQ = parallel LINQ, best for CPU-bound workloads
- TPL = general-purpose parallelism
- Combine wisely: pure LINQ core + parallelism at boundaries