

C# Interview Questions and Answers

Intern to senior level

C# and .NET interview questions and answers from beginner to senior: value vs reference types, string vs StringBuilder, const vs readonly, abstract class vs interface, LINQ, IEnumerable vs IQueryable, dependency injection, async/await and garbage collection.

12 questions Read online: <https://www.freedevtemplates.com/interview-questions/csharp>

1. What is the difference between a value type and a reference type in C#?

Level: intern C#

A value type variable holds its data directly and is copied on assignment, while a reference type variable holds a reference to an object, so assignment copies the reference and both variables see the same object.

Value types include int, double, bool, char, DateTime, enums and any struct. Reference types include class, string, arrays, delegates and object. A value type is stored inline where it is declared: on the stack for a local variable, or inside the containing object on the heap. A reference type's object always lives on the managed heap and is cleaned up by the garbage collector.

This changes behaviour when you pass values to methods. Passing a struct hands the method its own copy, so changes inside do not leak out. Passing a class passes a copy of the reference, so the method can modify the same object. Reference types can be null; value types cannot, unless you use `Nullable<T>` (int?).

Key points

Value types are copied, reference types share one object

Reference types can be null, value types need `int?` syntax

Large mutable structs are usually a bad idea

```
struct PointS { public int X; }
class PointC { public int X; }

var s1 = new PointS { X = 1 }; var s2 = s1; s2.X = 99;
Console.WriteLine(s1.X); // 1 (copy)

var c1 = new PointC { X = 1 }; var c2 = c1; c2.X = 99;
Console.WriteLine(c1.X); // 99 (same object)
```

2. What is the difference between string and StringBuilder in C#?

Level: intern C#

A string is immutable, so every change creates a new string object, while StringBuilder is a mutable buffer that modifies its contents in place, making it much faster for repeated concatenation.

Each time you do `s += "x"` in a loop, the runtime allocates a new string and copies the old content, which is $O(n^2)$ work overall and creates a lot of garbage. `StringBuilder` keeps an internal character buffer that grows as needed, so appending is cheap and you call `ToString()` once at the end.

For a handful of concatenations, a plain string or string interpolation is clearer and just as fast. Use `StringBuilder` when building text in a loop or from many pieces.

```
var sb = new StringBuilder();
for (int i = 0; i < 10_000; i++)
{
    sb.Append(i).Append(', ');
}
string result = sb.ToString();
```

3. What is the difference between const and readonly in C#?

Level: junior C#

A const is a compile-time constant that must be assigned where it is declared, while a readonly field can be assigned at declaration or in the constructor, so its value can be decided at runtime.

const values are baked into the calling code at compile time and are implicitly static. They can only be primitive types, strings or null. If you change a const in a library, every project using it must be recompiled to see the new value.

readonly fields are normal fields that cannot be reassigned after the constructor finishes. They can hold any type, can differ per instance, and are read at runtime. Note that readonly makes the reference unchangeable, not the object it points to. For compile-time values that will never change (such as the number of days in a week) use const; otherwise prefer readonly.

```
public class Config
{
    public const int MaxRetries = 3; // compile-time
    public readonly DateTime CreatedAt; // runtime, per instance

    public Config() { CreatedAt = DateTime.UtcNow; }
}
```

4. What is the difference between ref and out parameters in C#?

Level: junior C#

Both pass an argument by reference, but ref requires the variable to be initialised before the call, while out requires the method to assign a value before it returns.

Use ref when the method both reads and modifies the caller's variable. Use out when the method only produces a value, such as the bool TryParse(string, out int result) pattern, which lets a method return a success flag and a value together. A third modifier, in, passes by reference as read-only, useful to avoid copying large structs.

In modern C# you can often return a tuple instead ((bool ok, int value)), which is clearer than out parameters.

```
if (int.TryParse(input, out int number))
{
    Console.WriteLine(number * 2);
}

static void AddTen(ref int x) => x += 10;
int n = 5;
AddTen(ref n); // n is now 15
```

5. What is the difference between an abstract class and an interface in C#?

Level: junior C#

An abstract class can hold state, constructors and partial implementation but a class can inherit only one, while an interface defines a contract that a class can implement many of.

Use an abstract class when related classes share fields, constructor logic or common behaviour, and you want to force subclasses to fill in the rest. A class can extend only one base class, so this is a strong coupling.

Use an interface to describe a capability (IDisposable, IComparable, IEmailSender) that unrelated types can implement, and to make code testable and replaceable through dependency injection. Since C# 8 interfaces may include default method implementations, but they still cannot hold instance fields. A common guideline: start with an interface for the contract, and add an abstract base class only if there is real shared code.

Key points

Abstract class: shared state and code, single inheritance

Interface: contract, multiple implementation

Interfaces make dependency injection and testing easier

```
public interface INotifier { void Send(string message); }

public abstract class NotifierBase : INotifier
{
    protected readonly ILogger Logger;
    protected NotifierBase(ILogger logger) => Logger = logger;
    public abstract void Send(string message);
}
```

6. What is boxing and unboxing in C# and why does it affect performance?

Level: junior C#

Boxing converts a value type to object by allocating it on the heap, and unboxing converts it back; both add allocations and casts, so avoid them in hot code paths.

Assigning an int to an object variable boxes it: the runtime allocates a new object on the heap and copies the value in. Casting back with (int)obj is unboxing, which checks the type and copies the value out. Boxing in a loop creates garbage that the collector has to clean.

Generics were introduced to avoid this: List<int> stores ints directly, whereas the old ArrayList boxed every item. Watch for hidden boxing when a struct is used as an interface type, or when you pass a value type to a method taking object, such as older string.Format overloads.

```
int i = 42;
object boxed = i; // boxing: heap allocation
int j = (int)boxed; // unboxing: type check + copy

var list = new List<int>(); // no boxing
list.Add(i);
```

7. What is LINQ in C# and what is deferred execution?

Level: junior C#

LINQ (Language Integrated Query) lets you query collections, databases and XML with a consistent syntax, and deferred execution means a query does not run when you define it but when you enumerate the results.

Methods such as Where, Select and OrderBy only build up a description of the query. The work happens when you iterate with foreach or call an operator that forces evaluation, such as ToList(), ToArray(), Count() or First().

This is efficient, but it has traps: if the source changes between defining and enumerating the query, you see the new data, and enumerating the same query twice runs it twice (a database query would hit the database twice). Call ToList() when you want a snapshot or need to reuse the results.

```
var numbers = new List<int> { 1, 2, 3 };
var evens = numbers.Where(n => n % 2 == 0); // nothing runs yet

numbers.Add(4);
Console.WriteLine(string.Join(", ", evens)); // 2,4 (runs now)

var snapshot = numbers.Where(n => n > 1).ToList(); // runs immediately
```

8. What is the difference between IEnumerable and IQueryable in C#?

Level: mid C#

IEnumerable runs the query in memory using delegates, while IQueryable builds an expression tree that a provider such as Entity Framework translates into SQL, so filtering happens in the database.

If you call `.Where(...)` on an `IQueryable<Customer>` from Entity Framework, the condition becomes part of the SQL WHERE clause and only matching rows come over the network. If you call it on an `IEnumerable<Customer>`, all rows are loaded first and then filtered in your application's memory.

A common bug is converting to `IEnumerable` or calling `ToList()` too early, which silently loads the full table. Keep the query as `IQueryable` until you have applied all filtering, sorting and paging, and only then materialise it. Remember that an `IQueryable` can only contain operations the provider can translate to SQL.

Key points

`IQueryable`: expression trees translated to SQL

`IEnumerable`: in-memory iteration with delegates

Materialise (`ToList`) as late as possible

```
// SQL: SELECT ... WHERE IsActive = 1 (filtered in the database)
var active = dbContext.Customers.Where(c => c.IsActive).ToList();

// loads ALL customers first, then filters in memory
IEnumerable<Customer> all = dbContext.Customers.ToList();
var slow = all.Where(c => c.IsActive).ToList();
```

9. What are transient, scoped and singleton service lifetimes in ASP.NET Core dependency injection?

Level: mid C#

Transient services are created every time they are requested, scoped services are created once per HTTP request, and singleton services are created once and shared for the life of the application.

Register services in `Program.cs` with `AddTransient`, `AddScoped` or `AddSingleton`. Use transient for lightweight, stateless services, scoped for things tied to a request such as an Entity Framework `DbContext`, and singleton for shared, thread-safe objects such as caches and configuration.

The classic bug is a captive dependency: a singleton that depends on a scoped service keeps that one instance alive forever, so a `DbContext` ends up shared across requests and threads. ASP.NET Core can detect this in development with scope validation. Singletons must also be thread-safe because many requests use them at once.

Key points

Transient: new every time. Scoped: per request. Singleton: one for the app

Never inject a scoped service into a singleton

Singletons must be thread-safe

```
builder.Services.AddTransient<IEmailBuilder, EmailBuilder>();
builder.Services.AddScoped<AppDbContext>();
builder.Services.AddSingleton<IMemoryCache, MemoryCache>();

public class OrdersController(AppDbContext db, IEmailBuilder email) : ControllerBase { }
```

10. What is the difference between Task and Thread, and how does async/await work in C#?

Level: mid C#

A Thread is an operating-system execution unit, while a Task represents an asynchronous operation scheduled on the thread pool; await releases the current thread while the operation is pending and resumes the method when it completes.

Creating threads is expensive and you must manage them yourself. A Task is a higher-level abstraction: it can represent CPU work queued to the thread pool (Task.Run) or I/O that does not need a thread at all while waiting, such as an HTTP call or database query.

When the compiler sees await, it turns the method into a state machine. If the awaited task is not finished, the method returns control to its caller and the thread is free to serve other requests. When the task completes, the rest of the method continues, possibly on another thread. This is why async I/O lets a web server handle many more concurrent requests with the same number of threads. async/await is about scalability for I/O-bound work; for CPU-bound work it does not make anything faster by itself.

Key points

Task = promise of work, not a thread

await frees the thread during I/O

Use Task.WhenAll for independent operations

```
public async Task<string> GetPageAsync(HttpClient client, string url)
{
    // thread is released while waiting on the network
    string html = await client.GetStringAsync(url);
    return html.ToUpperInvariant();
}

// run independent calls concurrently
var results = await Task.WhenAll(GetPageAsync(c, a), GetPageAsync(c, b));
```

11. How does garbage collection work in C# and when should you use IDisposable?

Level: mid C#

The .NET garbage collector automatically frees managed objects that are no longer reachable, organising them into generations 0, 1 and 2 so short-lived objects are collected cheaply, and IDisposable is used to release unmanaged resources such as files and connections promptly.

New objects start in generation 0. Objects that survive a collection are promoted to generation 1 and then to generation 2. Collecting generation 0 is fast and frequent, while generation 2 collections are expensive. Objects of roughly 85,000 bytes or more go on the large object heap, which is collected with generation 2, so allocating many large temporary arrays can hurt performance.

The GC knows about memory, not about files, sockets or database connections. Types that hold those implement IDisposable, and you should wrap them in a using statement so the resource is released deterministically, instead of waiting for a finalizer. Common memory problems are not classic leaks but objects kept reachable by static fields, event handlers that are never unsubscribed, and caches without limits.

Key points

Generational GC: gen 0, 1, 2 and the large object heap
Dispose releases unmanaged resources; GC releases memory
Static references and event handlers keep objects alive

```
await using var connection = new SqlConnection(connectionString);
await connection.OpenAsync();
// connection is disposed (and returned to the pool) at the end of the scope
```

12. How do you avoid deadlocks with async/await in C#?

Level: senior C#

Use async all the way down instead of blocking on tasks with .Result or .Wait(), and in library code use ConfigureAwait(false) so continuations do not need to return to a captured context.

The classic deadlock happens in UI apps and legacy ASP.NET: the calling thread blocks on task.Result, while the awaited method is waiting to resume on that same captured synchronisation context. Neither can proceed. ASP.NET Core has no such context, so it avoids this particular deadlock, but blocking calls still waste threads and can cause thread-pool starvation under load.

Rules to follow: make the whole call chain async and await it; never call .Result, .Wait() or .GetAwaiter().GetResult() on a task that is not already complete; avoid async void except for event handlers; pass a CancellationToken through your calls; and use ConfigureAwait(false) in general-purpose libraries. Lock statements cannot contain await, so use SemaphoreSlim.WaitAsync() for async mutual exclusion.

Key points

Do not block on async code

Avoid async void

ConfigureAwait(false) in libraries, CancellationToken everywhere

```
// risky: blocks the calling thread
var data = GetDataAsync().Result;

// correct: async all the way
public async Task<IActionResult> Index(CancellationToken ct)
{
    var data = await GetDataAsync(ct);
    return Ok(data);
}
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/csharp>