C#

Version 14 · verified 2026-07-26

A gradual walk through C#'s type system, records, pattern matching, nullable refs, LINQ, and async/await.

Contents

Mental model

C# splits every type into two kinds: value types (`struct`, `int`, `bool`) that copy on assignment, and reference types (`class`) that share on assignment — most surprising bugs come from treating one like the other. Nullable reference types are a compile-time-only annotation: `string?` warns if you might dereference null, but nothing stops a real `null` from reaching that line at runtime. `await` is unrelated to threads by default — it frees the calling thread while waiting, it does not create a new one.

Types, variables & nullability

Syntax	Meaning
`var x = 5;`	inferred as `int`
`int? x = null;`	nullable value type
`string? s = null;`	nullable reference type
`const int Max = 10;`	compile-time literal
`readonly int id;`	set once, in constructor

`var` infers the type at compile time — it is still static typing, not `dynamic`. `readonly` differs from `const`: `const` is a literal baked into every caller at compile time, while `readonly` is resolved once per instance and can depend on runtime values.

```
int? age = null;
age ??= 30;
```

```
Console.WriteLine($"Age: {age}");
// Age: 30
```

> **Gotcha:** boxing an `int? x = 5` to `object` unwraps it — `x.GetType()` reports `System.Int32`, never `Nullable<int>`. The wrapper only exists while typed.

Classes, records & structs

Kind	Copy	Equality
`class`	reference	reference (default)
`record`	reference	by value (members)
`struct`	value	by value (fields)
`record struct`	value	by value (members)

A `record` gets compiler-generated value equality, `ToString()`, and a `with` expression for non-destructive copies, for free. Primary constructors put parameters directly in scope for use in the body.

```
var a = new Point(1, 2);
var b = a with { Y = 3 };
Console.WriteLine(a == b);
// False -- value equality, Y differs
Console.WriteLine(b);
// Point { X = 1, Y = 3 }
```

```
record Point(int X, int Y);
```

> **Gotcha:** `class Point(int X, int Y)` does not expose `X`/`Y` as properties like a record does — on a plain class, primary constructor parameters are only usable inside the class body unless you declare a property.

Interfaces & polymorphism

Feature	Meaning
`interface I { }`	a contract; implement many
default method body	body lives in the interface
`abstract class`	can also hold instance state
`virtual` / `override`	polymorphic dispatch
`sealed override`	stops further overriding

A class implements any number of interfaces but extends at most one base class. An interface can supply a default method body so adding a member does not break existing implementers — abstract classes can additionally hold shared instance state, which interfaces cannot.

```
interface IGreeter
{
  string Greet(string name) => $"Hi, {name}";
}
```

```
class Formal : IGreeter
{
  public string Greet(string name) =>
    $"Good day, {name}.";
}
```

Pattern matching & switch expressions

Pattern	Example
type pattern	`if (o is string s)`
property pattern	`{ Age: > 18 }`
relational pattern	`> 0 and < 100`
list pattern	`[first, .. ]`
switch expression	`x switch { ... }`

Pattern matching branches on shape, not just equality; an `is` pattern introduces a new variable only in the scope where it matched. A `switch` expression is itself an expression — every arm returns a value, and the compiler warns if the arms are not exhaustive.

```
Console.WriteLine(Describe(new Shape(4)));
// quadrilateral

string Describe(Shape s) => s switch
{
```

```
  { Sides: 3 } => "triangle",
  { Sides: 4 } => "quadrilateral",
  _ => "unknown",
};

record Shape(int Sides);
```

> **Gotcha:** `_` in a switch expression also matches `null`. Without an explicit `null =>` arm before it, a `null` input silently falls into the default case.

Nullable reference types

Syntax	Meaning
`string s`	compiler expects non-null
`string? s`	may be null
`s!`	null-forgiving, suppresses warning
`s?.Length`	null-conditional access
`s?.Prop = v`	null-conditional assignment

Nullable reference types (`<Nullable>enable</Nullable>` in the `.csproj`) are compile-time flow analysis only — they never insert a runtime check. `s!` genuinely disables that analysis rather than proving `s` is safe.

```
string? GetName() => null;
```

```
string? name = GetName();
Console.WriteLine(name!.Length);
// NullReferenceException at runtime
```

> **Gotcha:** the annotations are not part of the runtime type system. A library compiled without `Nullable` enabled looks entirely non-null to your code, even if it actually returns `null`.

Collections & LINQ

Type	Use
`List<T>`	resizable, indexable array
`Dictionary<K,V>`	hash map
`IEnumerable<T>`	lazy, forward-only sequence
`.Where` / `.Select`	filter / project, lazy
`.ToList()`	force evaluation now

LINQ method syntax composes lazily: `.Where` / `.Select` build a pipeline that runs nothing until you enumerate it with `foreach`, `.ToList()`, or `.Count()`.

```
List<int> nums = [1, 2, 3, 4, 5];
var evens = nums
  .Where(n => n % 2 == 0)
  .Select(n => n * n);
Console.WriteLine(
```

```
  string.Join(",", evens));
// 4,16
```

> **Gotcha:** a query variable is not a snapshot. If the source collection changes before you enumerate, the query reflects the new contents, not what existed when you wrote `.Where(...)`.

Async & await

Signature	Meaning
`async Task M()`	async, no return value
`async Task<T> M()`	async, returns `T`
`async void M()`	fire-and-forget, avoid
`await expr`	suspend, free the thread
`CancellationToken`	cooperative cancellation

`await` does not block the calling thread — it schedules a continuation and returns control immediately, which is why one thread can juggle thousands of in-flight `await`s. The method body runs synchronously up to the first genuinely incomplete `await`.

```
async Task<string> FetchAsync()
{
  await Task.Delay(100);
  return "done";
}
```

```
string result = await FetchAsync();
Console.WriteLine(result);
// done
```

> **Warning:** an exception inside `async void` has no caller `Task` to propagate to — it surfaces on the `SynchronizationContext` and usually crashes the process. Reserve `async void` for event handlers.

Exception handling

Construct	Use
`try/catch/finally`	handle; `finally` always runs
`catch (FooEx e) when (c)`	filtered catch
`using var x = ...;`	dispose at end of block
custom exception	extend `Exception`

Catch the most specific type you can actually handle — a bare `catch (Exception)` that only logs and rethrows adds little over not catching at all. A `using` declaration (no braces) disposes at the end of the enclosing block.

```
try
{
  throw new OutOfStockException("SKU1");
}
```

```
catch (OutOfStockException e)
{
  Console.WriteLine(e.Message);
}
// SKU1 is out of stock


class OutOfStockException(string sku)
  : Exception($"{sku} is out of stock");
```

> **Gotcha:** a `return` inside `finally` silently swallows any exception in flight, or the `try` block's own return value. Never `return` from `finally`.

Further reading

- C# language reference
- What's new in C# 14
- Nullable reference types
- LINQ overview