C# Intermediate/Advanced

A Complete 8-Chapter Programming Course

Topics covered:

Reified generics & constraints · LINQ, deferred execution & IQueryable

Delegates, events & the real lambda mechanism · async/await

Records & pattern matching · Nullable reference types

Extension methods & LINQ's secret · Capstone project

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed against Java, Kotlin, and TypeScript

Course 2 of 2 · completes the C# track

Table of Contents

CHAPTER 1 OF 8

Generics In Depth

COURSE 2 · CH 1

Generics In Depth

Reified generics — the exact opposite design choice from java2-1's own type erasure

Course 1 closed on a preview of LINQ. Course 2 opens on the mechanism underneath nearly everything else in C#'s type system — and the place where C# and Java's shared architecture diverges most sharply of all.

Generic Classes & Methods — Recap

```
class Box<T> {
    public T Value { get; set; }
}
static T First<T>(List<T> list) => list[0];
```

Same surface syntax as `java2-1`'s own generics — a type parameter `T`, filled in at each use site.

Reified Generics — Real Runtime Type Preservation

```
List<string> strings = new();
List<int> ints = new();

strings.GetType() == ints.GetType();  // false — genuinely DIFFERENT runtime types
```

Here's the central reversal: `java2-1` established that Java erases generic type information after compile-time checking, leaving `List<String>` and `List<Integer>` as the exact same runtime class. C# does the opposite — the CLR keeps real, distinct runtime type information for each type argument. `List<string>` and `List<int>` are genuinely different types at runtime, not just at compile time.

What Reification Actually Buys You

```
static T CreateDefault<T>() where T : new() {
    return new T();       // legal in C# — impossible in java2-1's Java
}

Console.WriteLine(typeof(T));  // also legal — typeof(T) inside a generic method
```

This resolves `java2-1`'s own "consequences of erasure" list one by one: `new T()` works (given a constraint, covered next), `typeof(T)` works directly, and generic arrays of `T` can be created without the reflection workarounds Java's own erased generics require.

Generic Constraints

```
class Repo<T> where T : class {}          // T must be a reference type
class Box<T> where T : struct {}          // T must be a value type
class Sorter<T> where T : IComparable<T> {}  // bounded, like java2-1's own bound
```

Similar spirit to `java2-1`'s own bounded type parameters — but with more constraint kinds available. The `class` and `struct` constraints have no Java equivalent at all, precisely because Java has no reference/value type split to constrain against in the first place, unlike `csharp1-2`'s own `struct`/`class` distinction.

Performance Implications of Reification

Here's the real payoff: `List<int>` in C# stores real `int` values directly, with zero boxing overhead — genuinely comparable to `cpp2-1`'s own C++ templates, not to `java1-8`'s forced-boxing `List<Integer>`. This directly resolves the exact gap `java1-8` named as a genuine cost of Java's erased generics — a real, measurable memory and performance difference for value-heavy collections.

The Real Cost — Code Bloat, Selectively

Reification isn't free, and it isn't identical to C++'s own approach either — worth stating honestly rather than glossing over. For **value types**, the CLR genuinely specializes a distinct compiled implementation per type, the same class of cost `cpp2-1`'s templates pay for every instantiation. But for **reference types**, the CLR is smarter: since every reference is the same pointer size underneath, the JIT can share one compiled implementation across *all* reference-type instantiations — `List<Cat>` and `List<Dog>` genuinely share compiled code, unlike `List<int>` and `List<double>`. This is a real, distinct middle ground between C++'s full monomorphization and Java's full erasure, not simply "closer to one or the other."

ASPECT	C++ TEMPLATES (CPP2-1)	JAVA (JAVA2-1)	C#
Strategy	full monomorphization	type erasure	reified — selective specialization
Value-type instantiations	separate compiled code, always	n/a — everything boxes	separate compiled code, like C++
Reference-type instantiations	separate compiled code, always	one shared implementation	one shared implementation, like Java
Runtime type info for T	n/a — resolved at compile time	none — erased	real — typeof(T) works

THE NEW() CONSTRAINT ENABLES REAL GENERIC FACTORIES

`where T : new()` lets a generic method construct `T` directly — exactly the capability `java2-1` 's own Challenge 2 demonstrated as flatly impossible in Java's erased generics.

REIFICATION DOESN'T MEAN EVERY INSTANTIATION GETS SEPARATE CODE

Don't assume `List<Cat>` and `List<Dog>` produce two fully separate JIT'd implementations the way `List<int>` and `List<double>` do — reference-type generic instantiations genuinely share compiled code under the hood. Reification's real benefit for reference types is preserved runtime type information, not full C++-style code duplication.

Coding Challenges

CHALLENGE 1

Create a List<string> and a List<int>, print whether their GetType() values are equal, and explain the result in a comment in terms of reification — contrasting it directly with java2-1's own erasure example.

View solution

CHALLENGE 2

Write a generic method CreateDefault<T>() constrained with where T : new() that returns new T(), and call it successfully with two different types that each have a public parameterless constructor.

View solution

CHALLENGE 3

Write a short paragraph, as a comment, explaining why List<Cat> and List<Dog> can share one compiled implementation under the hood while List<int> and List<double> cannot, tying your answer to the pointer-size argument from this chapter.

View solution

CHAPTER 1 QUICK REFERENCE

- C# generics are reified — real runtime type info preserved, the exact opposite of java2-1's own Java type erasure
- new T() and typeof(T) both work directly in C# — impossible under Java's erasure without workarounds
- Generic constraints include class/struct (no Java equivalent) alongside bounded constraints like java2-1's own
- Value-type generic instantiations get separate compiled code (like C++); reference-type instantiations share one implementation (like Java) — a genuine middle ground, not simply "closer to one side"
- List<int> stores real ints with zero boxing, resolving java1-8's own forced-boxing gap
- **Next chapter:** LINQ in depth — deferred execution, query syntax vs. method syntax, and IEnumerable<T> vs. IQueryable<T>

CHAPTER 2 OF 8

LINQ In Depth

COURSE 2 · CH 2

LINQ In Depth

Deferred, but genuinely re-enumerable — a real difference from java2-4's own single-use Streams

`csharp1-8` previewed LINQ's two syntaxes. This chapter goes underneath both — into what a LINQ query actually is before it runs, and where its behavior genuinely diverges from `java2-4`'s own Streams API.

Method Syntax vs. Query Syntax — When to Reach for Which

```
var result = people.Where(p => p.Age >= 18).OrderBy(p => p.Name);  // composes cleanly with anything els

var result2 = from p in people
              where p.Age >= 18
              orderby p.Name
              select p;  // reads naturally for joins and multi-source queries
```

Method syntax composes naturally with the rest of C# — custom extension methods, conditionals, anything else in scope. Query syntax tends to read more clearly once a query involves a `join` or spans multiple sources, where its SQL-like shape genuinely earns its keep.

Deferred Execution

```
var numbers = new List<int> { 1, 2, 3 };
var query = numbers.Where(n => n > 1);  // NOT executed yet — just describes the operation

numbers.Add(99);          // mutate the SOURCE after the query is defined

foreach (var n in query) Console.WriteLine(n);  // prints 2, 3, 99 — includes the late addition
```

Genuinely comparable to `java2-4`'s own lazy Streams — a LINQ query isn't run when it's defined, only when it's actually enumerated. But here's a real, concrete difference: a LINQ query can be **re-enumerated** — each `foreach`, each `.ToList()`, re-runs the whole query fresh against the *current* state of the source. `java2-4`'s

own Streams throw `IllegalStateException` on a second terminal operation; C#'s `query` above has no such restriction at all.

IEnumerable<T> vs. IQueryable<T>

```
IEnumerable<Customer> local = customers.Where(c => c.Active);       // runs as real C# delegates, in mem
IQueryable<Customer> remote = dbContext.Customers.Where(c => c.Active);  // becomes an EXPRESSION TREE
// remote is translated into real SQL and executed on the database — only when enumerated
```

`IEnumerable<T>` queries run as ordinary in-memory delegates, the same territory `java2-4`'s Streams live in entirely. `IQueryable<T>` — used by Entity Framework and similar tools — is a genuinely different mechanism: the same `.Where()` / `.OrderBy()` calls build an **expression tree** instead of executing anything, which gets translated into real SQL and run on a database server, only once the query is enumerated. `java2-4`'s Streams have no equivalent capability at all — they only ever operate over already-materialized in-memory Java objects.

A Deferred Execution Gotcha — Variable Capture

```
int threshold = 10;
var query = numbers.Where(n => n > threshold);  // captures threshold BY REFERENCE, not by value

threshold = 50;  // changed AFTER the query was defined

foreach (var n in query) Console.WriteLine(n);  // uses 50, not 10 — the live value at enumeration time
```

A real, opposite design choice from `java2-3`'s own Java lambdas: Java requires a captured local variable to be *effectively final* — reassigning it anywhere makes the capture illegal, a compile error. C# places no such restriction on ordinary local variables — the lambda captures the variable itself, not a snapshot of its value, so the query genuinely reads whatever `threshold` holds at the moment it's actually enumerated, not the moment it was written.

ASPECT	JAVA STREAMS (JAVA2-4 / JAVA2-3)	C# LINQ
Execution timing	deferred until a terminal op	deferred until enumeration
Re-use after first execution	throws IllegalStateException	fully re-enumerable, re-runs fresh each time
Remote/database translation	not available — in-memory only	IQueryable<T> — expression trees translated to SQL

ASPECT	JAVA STREAMS (JAVA2-4 / JAVA2-3)	C# LINQ
Captured local variable	must be effectively final	captured live, by reference — freely reassignable

FORCE IMMEDIATE EVALUATION WITH TOLIST()/TOARRAY() FOR A STABLE SNAPSHOT

When a query result needs to stay fixed regardless of later changes to the source or captured variables, materialize it immediately — `.ToList()` or `.ToArray()` runs the query once, right there, and hands back a real, independent collection.

ENUMERATING THE SAME QUERY TWICE CAN SILENTLY PRODUCE DIFFERENT RESULTS

Because a LINQ query re-runs fresh on every enumeration, two separate `foreach` loops over the same query variable can genuinely produce different output if the source collection or a captured variable changed in between — a real, easy-to-miss source of confusing bugs, especially coming from Java's single-use Stream model where this scenario simply can't arise.

Coding Challenges

CHALLENGE 1

Define a LINQ query over a List<int> filtering for values greater than 5, add a new qualifying element to the list after defining the query, then enumerate the query and show the new element is included.

View solution

CHALLENGE 2

Write a query capturing a local int variable in a Where() lambda, enumerate it once, then change the variable's value and enumerate the exact same query variable a second time, showing the two results differ.

View solution

CHALLENGE 3

Repeat Challenge 1's scenario, but call .ToList() immediately after defining the query. Add the same new element to the source list afterward, then print the materialized list, showing the new element is NOT included this time.

View solution

CHAPTER 2 QUICK REFERENCE

- Method syntax composes naturally; query syntax reads best for joins/multi-source queries
- LINQ queries are deferred, like java2-4's Streams — but genuinely re-enumerable, unlike Streams' single-use IllegalStateException restriction
- IEnumerable<T> runs in-memory like Java Streams; IQueryable<T> translates to expression trees and real SQL — no Java Streams equivalent exists
- C# lambdas capture local variables live, by reference — no effectively-final restriction, the opposite of java2-3's own Java rule
- .ToList()/.ToArray() force immediate evaluation for a stable, independent snapshot
- **Next chapter:** delegates, events, and lambda expressions — a different mechanism behind similar syntax to java2-3's own SAM interfaces

CHAPTER 3 OF 8

Delegates, Events & Lambda Expressions

COURSE 2 · CH 3

Delegates, Events & Lambda Expressions

Near-identical lambda syntax to java2-3 — hiding a genuinely different mechanism underneath

`csharp2-2`'s LINQ lambdas worked without ever asking what a lambda actually compiles to in C#. This chapter answers that — and the answer is a real, structural departure from `java2-3`'s own model.

Delegates — A Type-Safe Function Pointer

```
delegate int Calculator(int a, int b);

Calculator add = (a, b) => a + b;
Console.WriteLine(add(3, 4));  // 7
```

A `delegate` declares a type describing a method signature — genuinely similar in spirit to C's own function pointers, but type-checked at compile time the way a raw C function pointer never is. A `Calculator` variable can hold any method (or lambda) matching that exact signature.

Built-in Delegate Types — Func<> and Action<>

```
Func<int, int, int> add = (a, b) => a + b;   // last type parameter is the return type
Action<string> log = msg => Console.WriteLine(msg);   // void-returning
Predicate<int> isEven = n => n % 2 == 0;    // bool-returning, semantic name
```

Almost nobody declares a custom `delegate` type like `Calculator` in real code — `Func<T1, ..., TResult>` and `Action<T1, ...>` cover nearly every shape needed, genuinely comparable to `java2-3`'s own built-in `java.util.function` interfaces.

Lambdas Assigned to Delegates — The Real Mechanism

Here's the reveal, and it's a real divergence from `java2-3`: in Java, a lambda compiles to an anonymous implementation of a functional interface's single method — genuinely an OOP mechanism, an object with one

method. In C#, a lambda assigned to a delegate compiles to something different — a real compiler-generated method (often a private static method), and the delegate variable is a type-safe reference *to that method*, much closer in spirit to a function pointer than to an interface implementation. The syntax at the call site looks nearly identical in both languages; the mechanism underneath genuinely isn't.

Multicast Delegates

```
Action<string> notify = msg => Console.WriteLine("Log: " + msg);
notify += msg => Console.WriteLine("Email: " + msg);   // += adds a SECOND method to the same delegate

notify("Server started");  // invokes BOTH — "Log: ..." then "Email: ..."
```

A delegate variable can reference **multiple** methods at once via `+=`, invoking all of them in order when called — genuinely unique to C#, with no direct Java equivalent at all. This multicast capability is the exact mechanism the next feature is built on.

The event Keyword

```
public class Button {
    public event Action Clicked;   // a restricted multicast delegate

    public void Press() {
        Clicked?.Invoke();       // only Button itself can invoke it directly
    }
}

Button b = new Button();
b.Clicked += () => Console.WriteLine("Clicked!");  // outside code can only += / -=
b.Press();
```

`event` restricts a multicast delegate field: outside code can only `+=` / `-=` subscribers, never invoke it directly or reassign it with `=` — only the declaring class can. It's a genuine publish/subscribe pattern baked directly into the language, contrasted against Java's typical approach of hand-rolling a `List<Listener>` and manually iterating it to fire an event.

MECHANISM	C (C2-7)	JAVA (JAVA2-3)	C#
What it really is	a raw, untyped pointer to a function	an anonymous interface implementation	a type-safe reference to a method
Compile-time type checking	none — mismatched signatures compile	yes — via the functional interface	yes — via the delegate type

MECHANISM	C (C2-7)	JAVA (JAVA2-3)	C#
Referencing multiple functions at once	not built in	not built in	multicast delegates, built in (+=)
Built-in publish/subscribe	none	none — hand-rolled listener lists	event keyword

REACH FOR FUNC<>/ACTION<> BEFORE DECLARING A CUSTOM DELEGATE TYPE

A custom `delegate` declaration is worth it mainly when a genuinely descriptive name adds real clarity over `Func<int, int, int>` — otherwise the built-in types cover the shape just as well with less ceremony.

INVOKING AN EVENT WITH NO SUBSCRIBERS THROWS — GUARD IT

A multicast delegate or event with nobody subscribed is `null`, not an empty, safely-invokable list — calling it directly throws `NullReferenceException`. The `Clicked?.Invoke()` pattern, using `csharp1-3`'s own null-conditional operator, exists specifically to guard against this.

Coding Challenges

CHALLENGE 1

Declare a Func<int, int, int> that multiplies its two arguments and an Action<string> that prints a message, then call both and show the results.

View solution

CHALLENGE 2

Create an Action<string> delegate variable, add two separate lambdas to it using +=, then invoke it once and show both lambdas ran in order.

View solution

CHALLENGE 3

Write a class Alarm with a public event Action Triggered and a method Sound() that raises it safely using the null-conditional operator. Demonstrate calling Sound() both with and without a subscriber attached, showing neither call throws.

View solution

CHAPTER 3 QUICK REFERENCE

- A delegate is a type-safe method reference — compile-time checked, unlike c2-7's raw C function pointers
- Func<>/Action<>/Predicate<> cover nearly every shape, avoiding custom delegate declarations
- A C# lambda compiles to a real referenced method — a different mechanism from java2-3's anonymous interface implementation, despite near-identical call-site syntax
- Multicast delegates (+=) reference multiple methods at once, invoked in order — no Java equivalent
- event restricts a multicast delegate to +=/-= from outside code — a built-in publish/subscribe pattern Java hand-rolls instead
- Guard event invocation with ?.Invoke() — an unsubscribed event is null, not an empty list
- **Next chapter:** async/await — first-class language syntax since C# 5, contrasted against java2-5's own concurrency tools

CHAPTER 4 OF 8

Async/Await

COURSE 2 · CH 4

Async/Await

C# 5, 2012 — five years before the JavaScript syntax it directly inspired

`java2-5` covered concurrency the manual way — raw `Thread`, `synchronized`, thread pools. C# has a genuinely different, much higher-level answer for a large slice of that same territory, and it's the actual origin of syntax you may already recognize from JavaScript.

async/await Basics

```
async Task<string> FetchDataAsync() {
    await Task.Delay(1000);  // suspends here — doesn't block the thread
    return "data";
}

string result = await FetchDataAsync();
```

`async` marks a method as containing suspension points; `await` marks each one. C# 5 (2012) introduced this exact syntax pattern as a real language feature — genuinely the origin, not a parallel invention: JavaScript's own `async` / `await` (ES2017, 2017) was explicitly modeled on C#'s design, five years later.

Task and Task<T>

`Task` (no result) and `Task<T>` (a future result) are what an `async` method returns — conceptually comparable to a JS `Promise`. This is a genuinely higher-level tool than anything `java2-5` covered: `Thread` / `Runnable` / `synchronized` / `java.util.concurrent` are all real, manual thread-management primitives; `Task`-based async/await is a language feature built on top of that kind of machinery, not a replacement requiring the same manual bookkeeping.

What await Actually Does

`await` suspends the `async` method **without blocking the calling thread** — the thread is released to do other work, and the method resumes later via a continuation once the awaited `Task` completes. This is

genuinely comparable to JavaScript's own event-loop-based async model, not `java2-5`'s traditional blocking-thread model, where a thread sits idle (or is explicitly managed) while waiting.

async void — Fire-and-Forget, A Real Gotcha

```
async void OnButtonClick(object sender, EventArgs e) {  // only legal use case: event handlers
    await DoWorkAsync();
}

// async Task, not async void, for everything else — exceptions here CAN be caught normally
async Task ProcessAsync() { await DoWorkAsync(); }
```

`async void` exists specifically for event handlers, which can't return a `Task` due to the delegate signature they must match. A genuine, well-documented pitfall: an exception thrown inside an `async void` method can't be caught by its caller the normal way — it crashes the process instead of propagating as a catchable exception. Prefer `async Task` even for methods that logically return nothing.

Contrasted with java2-5's Concurrency Tools

`java2-5`'s `Thread`/`synchronized` model is about managing real OS threads directly — genuinely necessary for CPU-bound parallel work. C#'s `async`/`await` is about something different: not blocking a thread while waiting on I/O (network calls, disk, database queries) — for pure I/O-bound work, no extra thread is used at all while awaiting. For genuine CPU-bound parallel work, C# has `Task.Run()`, which does use the thread pool — that's the territory actually comparable to `java2-5`'s own material.

ASPECT	JAVA (JAVA2-5)	C#	JAVASCRIPT
Model	manual Thread/synchronized	async/await, language-level	async/await, language-level
Blocks the calling thread while waiting	yes, by default	no — thread released during await	no — single-threaded event loop
Syntax arrival	n/a — thread APIs since Java 1.0	C# 5 (2012) — the origin	ES2017 (2017) — modeled on C#'s

TASK.RUN() FOR CPU-BOUND WORK; PLAIN ASYNC/AWAIT FOR I/O-BOUND WORK

Reach for `Task.Run()` specifically when genuine parallel CPU work is needed — that's the territory closest to `java2-5`'s own thread-pool material. Plain `async`/`await` over I/O (network, disk, database) needs no extra thread at all while suspended.

AVOID ASYNC VOID OUTSIDE OF EVENT HANDLERS

An unhandled exception inside an `async void` method crashes the process rather than being catchable by the caller — a genuinely dangerous gotcha for code that looks completely ordinary at the call site. Use `async Task` everywhere except the one legitimate case: an event handler whose delegate signature requires `void`.

Coding Challenges

CHALLENGE 1

Write an async Task<int> method that awaits Task.Delay(500) and then returns 42, and call it with await from an async Main method, printing the result.

View solution

CHALLENGE 2

Write two async Task methods that each await a short delay and print a message, then call both without awaiting immediately (storing the returned Tasks), and finally await both together, demonstrating they ran concurrently rather than one after the other.

View solution

CHALLENGE 3

Write an async void method that throws an exception, call it, and show the caller cannot catch the exception with a normal try/catch around the call. Then rewrite it as async Task and show the exception CAN now be caught normally.

View solution

CHAPTER 4 QUICK REFERENCE

- async/await is real language syntax since C# 5 (2012) — the direct origin of JavaScript's own ES2017 async/await
- Task/Task<T> is what an async method returns — a much higher-level tool than java2-5's manual Thread/synchronized primitives
- await suspends without blocking the calling thread — the thread is released, resumed later via a continuation, closer to JS's event loop than Java's blocking-thread model
- async void is only for event handlers — exceptions inside it crash the process rather than being catchable; use async Task everywhere else
- Task.Run() is the genuine CPU-bound-parallelism tool, comparable to java2-5's own territory — plain async/await is for I/O-bound waiting, no extra thread needed

- **Next chapter:** records and pattern matching — C# 9 records, which genuinely shipped before java2-7's own Java 16 version

CHAPTER 5 OF 8

Records & Pattern Matching

COURSE 2 · CH 5

Records & Pattern Matching

C# 9 (2020) — a rare case where C# genuinely arrived before java2-7's own Java 16 records

`csharp1-4` previewed `init`-only properties as the mechanism this chapter builds on. This is where that promise pays off in full — real `record` syntax, and a capability neither Java's nor even Kotlin's own records quite match.

record — Full Syntax

```
public record Point(int X, int Y);  // one line — a positional record

Point p = new Point(3, 4);
p.X;                          // 3 — a real property, get; init;
p.ToString();                 // "Point { X = 3, Y = 4 }"
(int x, int y) = p;          // deconstruction — free, no extra code
```

A one-line positional `record` generates even more than `csharp1-4`'s own hand-assembled `init` properties: a constructor, `get; init;` properties for each component, correct `Equals()` / `GetHashCode()` / `ToString()`, and a `Deconstruct` method enabling tuple-style unpacking — all from one line.

Genuinely Comparable to java2-7's Records — But Arriving First

C# 9 (2020) shipped `record`; `java2-7`'s own Java records arrived in Java 16 (2021) — a rare case on this site where C# genuinely got there first, worth naming plainly given how often the comparison runs the other way. Both generate equality, hashing, and string formatting automatically. C#'s positional syntax is even more compact than Java's own record header syntax, which still requires listing each component explicitly in a similar but slightly more verbose form.

with-Expressions — Non-Destructive Mutation

```
Point p1 = new Point(3, 4);
Point p2 = p1 with { X = 99 };  // a NEW record — Y copied unchanged, X replaced

p1.X;  // still 3 — p1 is untouched
p2.X;  // 99
```

Here's the real reveal: `with` creates a brand-new record instance with just the named properties changed, copying everything else — something `java2-7`'s own Java records have no built-in equivalent for at all. Kotlin's `data class` is the actual match here, via its own `copy()` method — a case where C# converges with Kotlin rather than with Java.

record class vs. record struct

```
public record class Customer(string Name);   // reference type — the default, "record class" == "record"
public record struct Coordinate(double Lat, double Lng);  // value type (C# 10)
```

C# 10 let a record be either a reference type (`record class`, the plain `record` keyword's default) or a genuine value type (`record struct`) — layering the record's free boilerplate directly on top of `csharp1-2`'s own `struct`/`class` split. This is capability neither Java's nor Kotlin's records offer at all, since neither language has C#'s own value/reference type distinction to apply it to in the first place.

Pattern Matching, Building on csharp1-3's Light Touch

```
string Describe(Point p) => p switch {
    { X: 0, Y: 0 } => "origin",           // property pattern
    ( 0, _ ) => "on the Y axis",        // positional pattern — uses Deconstruct directly
    var pt when pt.X > 0 and pt.Y > 0 => "first quadrant",  // and-combinator
    _ => "elsewhere"
};
```

Property patterns (`{ X: 0, Y: 0 }`), positional patterns (using the record's own `Deconstruct` directly), and the `and`/`or`/`not` combinators are a genuinely rich pattern language — well beyond `java1-3`'s own simpler type-pattern-plus-switch-expression combination, and a natural extension of the type patterns `java1-3` already introduced.

ASPECT	JAVA RECORDS (JAVA2-7)	KOTLIN DATA CLASS	C# RECORD
Arrival	**Java 16 (2021)**	**Kotlin 1.0**	**C# 9 (2020) — first vs. Java**

ASPECT	JAVA RECORDS (JAVA2-7)	KOTLIN DATA CLASS	C# RECORD
Non-destructive copy with one field changed	`no built-in equivalent`	`copy() method`	`with { ... } expression`
Value-type option	`not applicable — Java has no value/reference split`	`not applicable`	`record struct (C# 10)`

USE WITH-EXPRESSIONS INSTEAD OF HAND-WRITING A NEW CONSTRUCTOR CALL

`p1 with { X = 99 }` reads directly as "the same record, except X" — cleaner and less error-prone than manually re-listing every unchanged property in a fresh constructor call.

WITH PERFORMS A SHALLOW COPY

A record containing a mutable reference-type field — a `List<T>`, for instance — shares that exact same list between the original and the `with`-modified copy. Mutating the list through either instance affects both, since `with` only copies the reference, not the referenced object itself.

Coding Challenges

CHALLENGE 1

Define a positional record Product(string Name, decimal Price), create an instance, print its ToString() output, and deconstruct it into two separate variables.

View solution

CHALLENGE 2

Using Challenge 1's Product record, create an instance, use a with-expression to produce a second instance with only the Price changed, and print both to show the original is unaffected.

View solution

CHALLENGE 3

Define a record Point(int X, int Y) and write a switch expression using a property pattern for the origin, a positional pattern for points on the X axis, and a discard for everything else. Test it against three different points.

View solution

CHAPTER 5 QUICK REFERENCE

- A one-line positional record generates a constructor, get;init; properties, Equals/GetHashCode/ToString, and Deconstruct
- C# records (C# 9, 2020) genuinely shipped before java2-7's own Java 16 (2021) records — a rare "C# first" case
- with-expressions create a modified copy with named properties changed — no Java record equivalent, but a real match with Kotlin's data class copy()
- record struct (C# 10) makes a record a value type — capability neither Java nor Kotlin records offer, since neither has C#'s value/reference split
- Property, positional, and and/or/not pattern combinators extend java1-3's own simpler type patterns considerably
- with is a shallow copy — mutable reference fields are shared between original and copy
- **Next chapter:** nullable reference types — a three-way comparison against Kotlin's built-in null safety and Java's total lack of either

CHAPTER 6 OF 8

Nullable Reference Types

COURSE 2 · CH 6

Nullable Reference Types

Opt-in and warning-only — genuinely less than Kotlin's real, enforced null safety, genuinely more than Java's nothing at all

This chapter is deliberately honest rather than flattering to C#: nullable reference types are a real, useful safety net, but they don't put C# on equal footing with Kotlin's own approach — and being clear about exactly where the line sits matters more than a tidy "C# has null safety too" claim.

The Problem — Null Reference Exceptions

Every reference type in C#, per `java1-2`'s own reference-type material, can normally hold `null` — dereferencing a null reference throws `NullReferenceException` at runtime, the same class of bug Java's own `NullPointerException` represents.

Nullable Reference Types — Opt-In, Not Opt-Out

```
// with nullable reference types enabled:
string name = null;       // compiler WARNING — string means "not null" now
string? nickname = null;  // fine — the ? explicitly allows null
```

C# 8 (2019) introduced nullable reference types as an **opt-in** feature — enabled via `#nullable enable` or project-wide in the `.csproj`. Once enabled, the compiler treats every reference type as non-nullable *by default*, requiring an explicit `?` (`string?`) to allow null — inverting the language's own 20-year-old default.

Kotlin's Built-In Null Safety, By Contrast

`kotlin1-3`'s own null safety was never retrofitted — `String` vs. `String?` has been the *only* way Kotlin has worked since Kotlin 1.0. There's no context to enable or disable; it's simply how the type system has always functioned. C#'s version is a compiler feature layered on top of two decades of a language that was nullable-everywhere before C# 8 ever existed.

Java's Total Lack of Either

`java1-2` / `java1-4` 's own Java has no built-in null-safety mechanism at all — every reference type is always nullable, with zero compiler warning for a potential null dereference, and no opt-in feature to change that. The honest three-way picture: Java offers no protection whatsoever; C# offers an opt-in warning system; Kotlin offers real, type-system-enforced protection from day one.

Nullable Reference Types Are Warnings, Not Guarantees

```
string? maybeNull = GetName();
Console.WriteLine(maybeNull.Length);  // WARNING — but this still compiles and can still throw!
```

The crucial honesty point: unlike Kotlin's real compile-time **errors** for unsafe null access, C#'s nullable reference type violations are, by default, only **warnings**. The code above still compiles cleanly and can still throw `NullReferenceException` at runtime if the warning is ignored, or if the static analysis is fooled — a value arriving from an un-annotated external library is a common real case. This is a genuine, real limitation, not an edge case to gloss over.

The Null-Forgiving Operator !

```
string? maybeNull = GetName();
Console.WriteLine(maybeNull!.Length);  // "trust me" — suppresses the warning, ZERO runtime check added
```

`!` tells the compiler "trust me, this isn't null here," suppressing the warning with no verification of any kind. It's a real escape hatch, and genuinely dangerous if the assertion turns out to be wrong — pure compiler-trust, backed by nothing at runtime.

ASPECT	JAVA	C#	KOTLIN (KOTLIN1-3)
Null-safety mechanism	none at all	nullable reference types (opt-in)	built into the type system
When introduced	never	C# 8 (2019), 20 years in	Kotlin 1.0, from the start
Violation severity	no warning, no error	warning only, by default	compile-time error
Escape hatch	n/a	! null-forgiving operator	!! not-null assertion (also unsafe)

ENABLE NULLABLE REFERENCE TYPES ON EVERY NEW C# PROJECT

Retrofitting nullable reference types onto an existing, large codebase is real work — but turning it on for a brand-new project costs nothing and starts every file with the safer, non-null-by-default posture from day one.

A NULLABLE WARNING IS NOT A COMPILE ERROR — THE BUILD STILL SUCCEEDS

Code with unaddressed nullable-reference warnings compiles and runs exactly as before, and can still throw `NullReferenceException` — genuinely different from Kotlin's real enforcement, where the equivalent code simply wouldn't compile at all. Don't treat "no warnings" as "provably null-safe" the way Kotlin's own guarantee actually is.

Coding Challenges

CHALLENGE 1

With nullable reference types enabled, write a method that assigns null to a plain string variable and show the resulting compiler warning, then fix it by changing the variable's type to string?.

View solution

CHALLENGE 2

Write a method that dereferences a string? that could genuinely be null, deliberately ignoring the compiler warning, and show that the program still compiles and then throws NullReferenceException at runtime when actually run with a null value.

View solution

CHALLENGE 3

Write a short paragraph, as a comment, explaining why Kotlin's null-safety violation is a compile-time error while the equivalent C# nullable-reference-type violation is only a warning, referencing kotlin1-3 directly.

View solution

CHAPTER 6 QUICK REFERENCE

- Nullable reference types (C# 8, 2019) are opt-in — enabled via #nullable enable or the .csproj, inverting the language's own long-standing nullable-by-default
- Kotlin's null safety (kotlin1-3) was never retrofitted — String vs. String? has been the only option since Kotlin 1.0
- Java has no null-safety mechanism at all — no warning, no error, ever

- C# nullable violations are warnings by default, not errors — the code still compiles and can still throw at runtime, genuinely different from Kotlin's real compile-time enforcement
- The ! null-forgiving operator suppresses a warning with zero runtime verification — a real, dangerous escape hatch if misused
- **Next chapter:** extension methods — the reveal that LINQ's .Where().Select() chaining is powered by them, a direct parallel to java2-3's own lambda reveal

CHAPTER 7 OF 8

Extension Methods — LINQ's Secret Revealed

COURSE 2 · CH 7

Extension Methods — LINQ's Secret Revealed

.Where() was never a method on your collection at all — a direct parallel to java2-3's own lambda reveal

`csharp1-8` and `csharp2-2` both used `.Where()` and `.Select()` without ever asking where those methods actually live. This chapter answers that — and it's a genuine, satisfying reveal.

Extension Methods — Syntax

```
public static class StringExtensions {
    public static bool IsPalindrome(this string s) {  // the "this" modifier on the FIRST parameter
        return s == new string(s.Reverse().ToArray());
    }
}

"racecar".IsPalindrome();  // called with dot-syntax, on a type you don't own or control
```

A `static` method in a `static` class, with `this` on its first parameter, becomes callable with ordinary dot-syntax on that parameter's type — even a sealed, built-in .NET type like `string` that could never otherwise be modified or subclassed.

The Reveal — .Where()/.Select() Chaining IS Extension Methods

```
// roughly what System.Linq.Enumerable actually contains:
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate) { /* ... */ }
```

Here's the payoff: `numbers.Where(n => n > 5)` was never calling a method defined on `List<T>` or on arrays at all. `Where` is a **static extension method** defined in `System.Linq.Enumerable`, taking `IEnumerable<T>` as its `this` parameter — genuinely comparable to `java2-3`'s own reveal that a lambda was always an anonymous interface implementation. Here, `.Where()` was always a static method call in disguise, dressed up in dot-syntax.

Why This Matters — Extending Types You Don't Own

`string`, `List<T>`, and any BCL or third-party type can gain new "methods" without modifying their source or subclassing them. This is exactly the LINQ scenario in practice: `IEnumerable<T>` is an interface, and an interface can't have new members added to it after release without breaking every existing implementer of it. Extension methods sidestep that entirely — adding functionality without touching the interface contract at all.

Extension Methods vs. Kotlin's Extension Functions

This is genuinely the closest match on this site: Kotlin's own `fun String.isPalindrome(): Boolean` syntax is nearly identical in spirit, just attaching the extension directly to the type name rather than via a `this`-modified parameter. Both are the exact same underlying mechanism — compile-time-only syntax sugar over a static call, with no real modification to the extended type in either language.

A Real Limitation — No Access to Private Members

An extension method, however it's called, only ever gets what the type's public (or otherwise accessible) surface already exposes. This is genuinely **not** the same as adding a real method to the class — a true instance method can reach private fields; an extension method never can, no matter how it's written or called.

ASPECT	JAVA	KOTLIN	C#
Mechanism	`static utility methods only`	`extension functions`	`extension methods`
Call-site syntax	`Utils.isPalindrome(str) — no dot-sugar`	`str.isPalindrome() — real dot-syntax`	`str.IsPalindrome() — real dot-syntax`
Access to private members	`n/a`	`no`	`no`

REACH FOR EXTENSION METHODS ON TYPES YOU DON'T OWN

Extension methods are the right tool specifically for adding a method-feeling utility to a BCL type or a third-party library type — reach for a plain static helper class instead when the type in question is your own and could just get a real instance method.

A REAL INSTANCE METHOD ALWAYS WINS OVER AN EXTENSION METHOD WITH THE SAME SIGNATURE

If a type later adds a genuine instance method matching an existing extension method's name and signature, the instance method takes priority at every call site — the extension method is silently shadowed, not an error, but a real subtlety worth knowing about ahead of time.

Coding Challenges

CHALLENGE 1

Write an extension method WordCount() on string that returns the number of whitespace-separated words, and call it with dot-syntax on a string literal.

View solution

CHALLENGE 2

Write your own generic extension method MyWhere<T>(this IEnumerable<T> source, Func<T, bool> predicate) that reimplements basic filtering using yield return, and use it on a List<int> with the same dot-syntax LINQ's real Where() uses.

View solution

CHALLENGE 3

Write a class with a private field and attempt to write an extension method that reads that private field directly. Show the resulting compile error and explain why in terms of what extension methods can and cannot access.

View solution

CHAPTER 7 QUICK REFERENCE

- A static method with this on its first parameter becomes callable with dot-syntax on that parameter's type
- LINQ's .Where()/.Select() are extension methods on IEnumerable<T>, defined in System.Linq.Enumerable — never real methods on List<T> or arrays
- Extension methods add functionality to types (including interfaces) you don't own, without breaking existing implementers
- Genuinely the closest match to Kotlin's own extension functions — same compile-time-only mechanism, near-identical call-site syntax
- Extension methods never have access to private members — a real instance method always wins if one exists with the same signature
- **Next chapter:** the capstone — a real C# project combining OOP, generics, LINQ, async/await, and records

CHAPTER 8 OF 8

Capstone: Building a Small Project

COURSE 2 · CH 8 — CAPSTONE

Building a Small Project

A small library loan tracker combining nearly every chapter across both C# courses

Sixteen chapters, two courses — this capstone builds one small, real library loan tracker touching almost all of it: records for the data model, a generic repository queried with LINQ, an interface with a default method, an extension method, an async availability check, and an event for overdue notifications.

Designing the Data Model

```
public record Book(string Isbn, string Title, int CopiesAvailable);  // csharp2-5's positional record
public record Loan(string Isbn, string Borrower, DateTime DueDate);
```

`Book` and `Loan` are `csharp2-5` records — one line each generates the constructor, equality, and formatting the whole rest of this capstone relies on.

A Custom Exception

```
public class BookNotAvailableException : Exception {  // csharp1-7 — unchecked, like every C# exception
    public BookNotAvailableException(string isbn)
        : base($"No copies of book {isbn} are currently available.") {}
}
```

A plain `Exception` subclass — `csharp1-7` established there's no checked/unchecked split to navigate here, unlike Java's own capstone.

A Generic Repository, Queried with LINQ

```
public class Repository<T> {  // csharp2-1's reified generics
    private readonly List<T> _items = new();
    public void Add(T item) => _items.Add(item);
    public IEnumerable<T> All() => _items;
    public IEnumerable<T> Where(Func<T, bool> predicate) => _items.Where(predicate);  // csharp2-2's LIN
}
```

One `Repository<T>` class serves both `Repository<Book>` and `Repository<Loan>`. Its own `Where` method just forwards to LINQ's real `.Where()` — `csharp2-7`'s own reveal, still an extension method underneath, even when wrapped in a method that merely delegates to it.

Interfaces & Default Methods

```
public interface ILoanNotifier {  // csharp1-6
    void Notify(string message);
    void NotifyOverdue(Loan loan) => Notify($"Overdue: {loan.Borrower}, due {loan.DueDate:d}");  // defa
}
```

Extension Methods for Convenience

```
public static class LoanExtensions {  // csharp2-7
    public static bool IsOverdue(this Loan loan) => DateTime.Now > loan.DueDate;
}
```

`loan.IsOverdue()` reads like a real instance method, but `Loan` is an immutable record — this extension method adds behavior without ever touching the type itself.

Async Availability Checks

```
public static async Task<bool> CheckAvailabilityAsync(Repository<Book> books, string isbn) {  // csharp2
    await Task.Delay(200);  // simulating a remote inventory check
    var book = books.Where(b => b.Isbn == isbn).FirstOrDefault();
    return book is { CopiesAvailable: > 0 };  // csharp2-5's property pattern
}
```

Events for Overdue Notifications

```
public class LoanDesk {
    public event Action<Loan>? LoanOverdue;  // csharp2-3's event + csharp2-6's nullable reference type

    public void CheckIn(Loan loan) {
        if (loan.IsOverdue())
            LoanOverdue?.Invoke(loan);  // csharp1-3's ?. guard against a null (unsubscribed) event
    }
}
```

Putting It Together — A Small Run

```
var books = new Repository<Book>();
books.Add(new Book("001", "C# in Depth", 2));

if (!await CheckAvailabilityAsync(books, "001")) {
    throw new BookNotAvailableException("001");
}

var desk = new LoanDesk();
desk.LoanOverdue += loan => Console.WriteLine($"Reminder needed for {loan.Borrower}");

var loan = new Loan("001", "Alice", DateTime.Now.AddDays(-3));
desk.CheckIn(loan);  // prints "Reminder needed for Alice" — the loan is overdue
```

Chapter Attribution

CAPSTONE PIECE	CHAPTER
Book / Loan records	csharp2-5
BookNotAvailableException (unchecked)	csharp1-7
Repository<T> generics	csharp2-1
Repository.Where() forwarding to LINQ	csharp2-2, csharp2-7
ILoanNotifier interface & default method	csharp1-6
IsOverdue() extension method	csharp2-7
CheckAvailabilityAsync + property pattern	csharp2-4, csharp2-5
event LoanOverdue + ?.Invoke() guard	csharp2-3, csharp1-3, csharp2-6

What's Still Out of Scope

Honestly: `Repository<T>`'s internal `List<T>` isn't thread-safe — `csharp2-4`'s own async work never guaranteed thread-safety, only non-blocking suspension, and this capstone has no genuine concurrent access to worry about. No real persistence — everything is in-memory. No real network call — `CheckAvailabilityAsync` simulates one with `Task.Delay`. No automated tests. This capstone proves the pieces fit together, not that the result is production-ready.

A CAPSTONE'S VALUE IS IN THE SEAMS, NOT THE SIZE

This project is deliberately small — the point was never scale, it was confirming that records, generics, LINQ, events, and async/await genuinely compose cleanly together in real code, not just in isolated chapter examples.

THIS IS A TEACHING CAPSTONE, NOT A TEMPLATE FOR A REAL SYSTEM

A real loan-tracking system would need persistence, a real inventory API, concurrency-safe storage, and tests — treat this as proof the language features fit together, not as production-ready code to copy.

Coding Challenges

CHALLENGE 1

Add a Return(Loan loan) method to Repository<Loan>-backed code that removes a loan using LINQ to find it first, and demonstrate it removing the correct loan from a repository holding several.

View solution

CHALLENGE 2

Write a LINQ query over a Repository<Loan> that returns only overdue loans, using the IsOverdue() extension method inside the Where() predicate, and print the results using a record's own ToString().

View solution

CHALLENGE 3

Write a short paragraph (as a comment) explaining what would need to change in this capstone's Repository<T> to make it safe for concurrent access from multiple threads, referencing java2-8's own equivalent challenge and explaining any real differences the C# tools bring.

View solution

CHAPTER 8 QUICK REFERENCE — C# TRACK COMPLETE

- Records model the domain with free equality, formatting, and deconstruction (csharp2-5)
- A generic Repository<T> serves every entity type, backed by real reified generics (csharp2-1)
- LINQ queries the repository declaratively (csharp2-2), itself powered by extension methods (csharp2-7)
- Interfaces with default methods and a genuinely unchecked custom exception round out the OOP core (csharp1-6, csharp1-7)
- async/await handles simulated I/O without blocking (csharp2-4); events with a nullable-safe ?.Invoke() guard handle notifications (csharp2-3, csharp2-6)
- Concurrency safety, persistence, and testing are honestly named as still out of scope
- **Both C# courses are now complete — 16 chapters total, framed against Java, Kotlin, and TypeScript throughout.**