C# Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

The CLR & dotnet CLI · Value types, struct, and real property syntax
switch expressions & null-conditional operators · virtual/override dispatch
Interfaces & explicit interface implementation · Unchecked-only exceptions
Collections & a first taste of LINQ

Exercises: 24 hands-on exercises with worked solutions

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

Course 1 of 2 · Intermediate/Advanced follows

Table of Contents

CHAPTER 1 OF 8

Getting Started

COURSE 1 · CH 1

Getting Started

Same two-stage compilation model as java1-1's JVM — C# was built as Microsoft's direct answer to Java

This course sits deliberately next to Java throughout — not because the two are interchangeable, but because they share more architecture than most language pairs on this site, and diverge in specific, namable places. This chapter starts with what they share.

The CLR vs. the JVM

C# compiles to **IL** (Intermediate Language) — not directly to machine code — and the **CLR** (Common Language Runtime) JIT-compiles that IL at runtime, exactly the same two-stage model `java1-1` described for the JVM and Java bytecode. This is a genuine point of similarity worth naming explicitly, not a coincidence — C# inherited this architecture on purpose, the same "write once, run wherever the runtime exists" tradeoff Java made first.

The dotnet CLI

```
dotnet new console -o MyApp   # scaffolds a new project
dotnet run                    # compiles AND runs in one command
dotnet build                  # compiles only, without running
```

Where `java1-1` used two separate tools — `javac` to compile, `java` to run — `dotnet` is one unified CLI covering project scaffolding, building, and running. `dotnet run` alone does what `javac` + `java` together did.

Top-Level Statements

```
// Program.cs — the entire program, no class or Main required
Console.WriteLine("Hello, C#!");
```

Since C# 9, a file can skip the class-and-`Main` boilerplate entirely for the simplest programs — a real, direct contrast to `java1-1`'s strict "everything lives in a class, with an explicit `public static void Main`" requirement. Under the hood, though, the compiler still generates that exact class-and-`Main` shape automatically — this is real syntactic sugar over the same model, not a fundamentally different one.

What's Really There

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Hello, C#!");  // what the top-level version compiles down to
    }
}
```

This is the explicit, traditional form — the same shape `java1-1` required from the very first line. Larger, real applications generally still write `Main` explicitly; top-level statements are mainly a convenience for small programs, scripts, and quick experiments.

C#'s Origin as Microsoft's Direct Answer to Java

C# was created in 2000, led by **Anders Hejlsberg** — previously the architect behind Turbo Pascal and Delphi, later the designer of TypeScript. Its creation followed directly from a Sun Microsystems lawsuit over Microsoft's own non-compliant Java implementation, J++ — rather than continue fighting over Java itself, Microsoft built its own language and runtime from scratch. This is real, documented history, not just a marketing framing — and it's exactly why C# and Java's architectures line up as closely as they do.

ASPECT	JAVA (JAVA1-1)	C#
Runtime	`JVM`	`CLR`
Intermediate format	`bytecode`	`IL`
Compile & run tooling	`javac, then java — two tools`	`dotnet — one unified CLI`
Minimum program shape	`always an explicit class + main`	`top-level statements allowed (C# 9+)`

TOP-LEVEL STATEMENTS SUIT SCRIPTS, NOT NECESSARILY LARGE APPS

Reach for top-level statements for small programs, quick experiments, and this course's own early examples — real multi-file applications generally still write `Main` explicitly once there's more than one entry point's worth of setup to reason about.

ONLY ONE FILE PER PROJECT MAY USE TOP-LEVEL STATEMENTS

A project can have exactly one file containing top-level statements — attempting a second produces a real compile error, since the compiler can only generate one implicit `Main` per project.

Coding Challenges

CHALLENGE 1

Create a new console project with dotnet new console, write a top-level-statement Program.cs that prints two lines of your choosing, and run it with dotnet run.

View solution

CHALLENGE 2

Rewrite Challenge 1's program using the explicit class Program with a static void Main(string[] args) method instead of top-level statements, and confirm it produces identical output.

View solution

CHALLENGE 3

Explain in a comment why C# and Java both use a two-stage compile-then-JIT model rather than compiling directly to native machine code, tying your answer to C#'s own origin as a direct answer to Java.

View solution

CHAPTER 1 QUICK REFERENCE

- The CLR JIT-compiles IL at runtime — the same two-stage model as the JVM and Java bytecode (java1-1)
- dotnet is one unified CLI for scaffolding, building, and running — Java needed javac and java separately
- Top-level statements (C# 9+) skip the class/Main boilerplate for simple programs — real sugar, not a different model underneath
- Only one file per project may use top-level statements
- C# was created in 2000 as Microsoft's direct answer to Java, led by Anders Hejlsberg — later also the designer of TypeScript
- **Next chapter:** variables and basic types — value types vs. reference types, and var type inference

CHAPTER 2 OF 8

Variables & Basic Types

COURSE 1 · CH 2

Variables & Basic Types

Java draws the primitive/reference line for you — C# hands you the pen

`java1-2` drew a hard line: eight built-in primitives, everything else a reference type, no way for a programmer to add to either side. C# keeps the same two-category split — but genuinely lets you choose which side a new type belongs on.

Value Types vs. Reference Types

A value type holds its data directly — assigning or passing it copies the actual data. A reference type holds a reference to data stored elsewhere — assigning or passing it copies the reference, not the underlying data. This is C#'s own version of `java1-2`'s primitive-vs-reference split, but with a genuine structural difference: in Java, only the eight built-in primitives are ever value-like — every programmer-defined type is automatically a reference type, no exceptions. In C#, a programmer can define a brand-new value type of their own.

struct vs. class

```
struct Point {   // a value type — copied by value
    public int X, Y;
}

class Person {  // a reference type — copied by reference, same as every java1-4 class
    public string Name;
}
```

`struct` declares a value type; `class` declares a reference type — otherwise, the two look almost identical to write. This is the real, structural choice `java1-4`'s own classes never offered: in Java, every custom type is unconditionally a reference type, full stop.

Built-in Types Are Just Structs

```
int x = 5;
x.ToString();       // legal — int genuinely IS a struct (System.Int32), a real object underneath
5.ToString();      // also legal, for the same reason
```

Here's the real reveal: `int`, `double`, and `bool` aren't magic keywords divorced from the type system — they're built-in aliases for real structs (`System.Int32`, `System.Double`, `System.Boolean`). Because they're genuinely structs, they can call methods directly, no wrapping required. This is a real, structural contrast with `java1-2`'s own primitives, which aren't objects at all — Java's `int` can never call a method on itself; it must first be boxed into an `Integer`.

var Type Inference

```
var count = 5;          // inferred as int at compile time
var name = "Alice";     // inferred as string at compile time
// count = "text";        // still a compile error — var is NOT dynamic typing
```

`var` lets the compiler infer a variable's type from its initializer — the type is fixed permanently at compile time, exactly as if it had been spelled out explicitly. This is genuinely still static typing, in the same spirit as C++'s `auto`, not a loophole into dynamic typing.

Nullable Value Types

```
int count = null;    // compile error — a plain value type can never be null
int? count = null;   // legal — opts in via Nullable<int> underneath
```

A value type genuinely can't be `null` by default — it's real data sitting directly in the variable, not a reference that could point to nothing. The `?` suffix opts a value type into nullability through `Nullable<T>`, a wrapper struct. This is a separate, earlier feature from Course 2's own **nullable reference types** chapter, which addresses the opposite problem — reference types, which could always be null by default until that later feature opted them out of it.

ASPECT	JAVA (JAVA1-2)	C#
Who can define a value type	nobody — only 8 built-ins	any programmer, via struct
Is int a real object?	no — must box to Integer first	yes — int genuinely is System.Int32
Calling a method on a literal	not possible directly	5.ToString() works directly

ASPECT	JAVA (JAVA1-2)	C#
Can a value type be null	`n/a — primitives have no null concept`	`only if explicitly marked nullable (int?)`

REACH FOR STRUCT FOR SMALL, IMMUTABLE, FREQUENTLY-COPIED DATA

A small value type like a coordinate pair avoids heap allocation and garbage-collector pressure entirely — a genuine performance-relevant choice, not just a style preference, when a type is small and copied often.

A LARGE STRUCT COPIED OFTEN CAN HURT, NOT HELP, PERFORMANCE

Value-type semantics mean a struct is copied in full on every assignment and every parameter pass — for a large struct, that copying cost can exceed the cost of copying a reference to a class instance. struct is a genuine tradeoff, not a free performance win in every case.

Coding Challenges

CHALLENGE 1

Write a struct Point with X and Y int fields, and a class Wallet with a decimal Balance field, create one instance of each, assign each to a second variable, modify the second variable's fields, and print both original and copy to show which one changed and why.

View solution

CHALLENGE 2

Write code that calls a method directly on an int literal (e.g. 42.ToString()) and explain in a comment why this is legal in C# but would not be legal on a raw int in Java without boxing.

View solution

CHALLENGE 3

Declare an int? variable, assign it null, then attempt the same with a plain int. Show the resulting compile error on the plain int and explain the difference in terms of value-type semantics.

View solution

CHAPTER 2 QUICK REFERENCE

- struct declares a value type (copied by value); class declares a reference type (copied by reference) — unlike Java, where only 8 built-ins are ever value-like
- C#'s int/double/bool are real structs (System.Int32, etc.) — they can call methods directly, unlike Java's non-object primitives
- var infers a type at compile time — still fully static typing, not dynamic typing
- A plain value type can never be null; int? opts in via the Nullable<T> wrapper struct
- struct is a genuine tradeoff — great for small, immutable, frequently-copied data, costly for large types copied often
- **Next chapter:** operators and control flow — switch expressions with pattern matching from day one

CHAPTER 3 OF 8

Operators & Control Flow

COURSE 1 · CH 3

Operators & Control Flow

Fallthrough forbidden by default — the opposite of C and Java's own switch default

Most of C#'s operators are shared syntax with every C-family language already covered on this site. The genuine differences are concentrated in one place: `switch`, and two operators Java has no direct equivalent for at all.

Standard Operators & Integer Division

Arithmetic, relational, and logical operators behave exactly as in C/C++/Java. Integer division still truncates toward zero — the same behavior `c1-3` and `java1-3` already covered, inherited unchanged.

switch Statements — Fallthrough Forbidden by Default

```
switch (day) {
    case 1:
        Console.WriteLine("Monday");
        // no break here — this is a COMPILE ERROR in C#, not a silent bug
    case 2:
        Console.WriteLine("Tuesday");
        break;
}
```

This is a genuine, real difference worth stating plainly: C#'s `switch` statement **forbids** implicit fallthrough between non-empty case blocks — the exact opposite default of `c1-3`'s C and `java1-3`'s Java. A case with a statement body must end in `break`, `return`, `throw`, or an explicit `goto case` — omitting all four is a compile error, not a bug waiting to happen at runtime. Stacked *empty* case labels (`case 1: case 2: DoSomething(); break;`) are still fine, since neither label has its own body to fall out of.

switch Expressions — Pattern Matching Since Day One

```
string result = day switch {
    1 or 2 or 3 => "Early week",
    6 or 7 => "Weekend",
    _ => "Other"  // the discard pattern — C#'s default case
};
```

C# 8 (2019) shipped `switch` expressions with real pattern matching already built in — genuinely before `java1-3`'s own arrow-based switch expressions landed in Java 14 (2020). Another "C# arrived first" moment, alongside Course 2's own records chapter.

Pattern Matching in switch Expressions

```
string Describe(object obj) => obj switch {
    int n when n > 0 => "a positive int",   // type pattern + relational guard
    string s when s.Length == 0 => "an empty string",
    null => "null",
    _ => "something else"
};
```

Type patterns, relational guards via `when`, and a dedicated `null` pattern were all present from C# 8's initial launch — a notably richer starting point than `java1-3`'s own simpler initial arrow-only form.

Null-Conditional and Null-Coalescing Operators

```
string city = customer?.Address?.City;   // ?. — short-circuits to null the instant anything is null
string display = city ?? "Unknown";  // ?? — supplies a default only if the left side is null
```

`?.` (null-conditional) short-circuits an entire chain to `null` the moment any link is `null`, instead of throwing. `??` (null-coalescing) supplies a fallback value only when the left-hand side is `null`. Neither operator has a direct Java equivalent — Java requires either manual null checks or an `Optional`-based rewrite to express the same idea.

FEATURE	C (C1-3)	JAVA (JAVA1-3)	C#
switch statement fallthrough	allowed by default	allowed by default	forbidden — compile error
switch expression arrival	n/a	Java 14 (2020)	C# 8 (2019) — first
Null-safe chaining operator	none	none — manual checks or Optional	?. and ??

CHAIN ?. AND ?? TOGETHER FOR CONCISE NULL-SAFE READS

`customer?.Address?.City ?? "Unknown"` reads as one concise, self-documenting expression — short-circuit through any null link, then supply a fallback — replacing several lines of nested null checks in one line.

C# CATCHES THE MISSING-BREAK BUG AT COMPILE TIME, NOT RUNTIME

Where `c1-3` and `java1-3` both warned that a missing `break` silently falls through into the next case, C#'s compiler simply refuses to build code shaped like that in the first place — a real, structural safety improvement over both languages' own default.

Coding Challenges

CHALLENGE 1

Write a switch statement over an int month (1-12) with a case that deliberately omits a break after a non-empty body. Show the resulting compile error, then fix it.

View solution

CHALLENGE 2

Write a switch expression over an object using type patterns to distinguish int, string, and null, each returning a different descriptive string, with a discard pattern as the final case.

View solution

CHALLENGE 3

Write a class hierarchy of at least two levels deep (e.g. a Customer with a nullable Address property, which itself has a nullable City property), and use ?. chained with ?? to safely read the city with a default fallback, without any explicit null checks.

View solution

CHAPTER 3 QUICK REFERENCE

- switch statements forbid implicit fallthrough between non-empty cases — a compile error, the opposite default of C (c1-3) and Java (java1-3)
- switch expressions with pattern matching shipped in C# 8 (2019), before Java's own arrow-based version in Java 14 (2020)

- Type patterns, relational when guards, and a null pattern were all present in C#'s switch expressions from the start
- ?. (null-conditional) and ?? (null-coalescing) have no direct Java equivalent — chain them for concise null-safe reads
- **Next chapter:** classes and objects — auto-implemented properties as a first-class language feature

CHAPTER 4 OF 8

Classes & Objects

COURSE 1 · CH 4

Classes & Objects

Properties are a real language feature here, not a naming convention java1-4 left to discipline

Constructors and fields work almost identically to `java1-4`. The real divergence is in how C# handles the getter/setter pattern every Java class relies on — it's not a pattern here at all.

Constructors — Quick Recap

```
class Account {
    private string _owner;
    public Account(string owner) { _owner = owner; }  // same shape as java1-4
}
```

Same idea, same shape as `java1-4`'s own constructors — a method sharing the class's name, no return type, run once via `new`.

Properties — A First-Class Language Feature

```
public class Customer {
    public string Name { get; set; }  // auto-implemented property — one line
}

Customer c = new Customer();
c.Name = "Alice";   // looks like direct field access...
Console.WriteLine(c.Name);  // ...but is really calling get/set underneath
```

In Java, "a property" is purely a *naming convention* — a private field plus a hand-written `getName()` / `setName()` pair, entirely a matter of programmer discipline that `java1-4`'s own language rules never enforce or even recognize. C# has real property syntax built directly into the language: `{ get; set; }` alone generates a hidden backing field and both accessors automatically — one line replaces what Java requires spelling out as three separate members.

Full Property Syntax with Custom Logic

```
private string _name;
public string Name {
    get => _name;
    set => _name = value.Trim();  // real logic, still called like plain field access
}
```

A property can run genuine logic in its accessors — validation, transformation, computed values — while the call site still reads exactly like ordinary field access: `customer.Name = " Alice ";` silently runs the trim logic. This is a real, different call-site ergonomic from Java's explicit `setName(...)` method call, which always visibly announces itself as a method call.

Read-Only and Init-Only Properties

```
public string Id { get; }             // read-only — settable only inside the constructor
public string Sku { get; init; }       // init-only (C# 9) — settable once, at construction

var product = new Product { Sku = "A1" };  // object-initializer syntax works with init
```

A get-only property can only be assigned inside the constructor. `init` (C# 9) is slightly more flexible — it allows object-initializer syntax to set the property exactly once, at construction time, then locks it — the same mechanism Course 2's own records chapter builds on internally for its immutable, boilerplate-free data carriers.

Access Modifiers, Revisited

C# has the same `public` / `private` / `protected` plus a genuine fourth level — but a structurally different one than `java1-4`'s package-private. C#'s `internal` means visible anywhere within the same **assembly** (a compiled project/DLL), a unit defined by how the project is built, not by folder or namespace structure the way Java's package-private is tied to the package a file physically sits in.

ASPECT	JAVA (JAVA1-4)	C#
Getter/setter pattern	`a naming convention only`	`real language syntax — get/set`
Call site for reading a value	`customer.getName() — visibly a method call`	`customer.Name — looks like field access`
The "fourth" access level	`package-private (no modifier) — folder/package-scoped`	`internal — assembly-scoped`

ASPECT	JAVA (JAVA1-4)	C#
Immutable-after-construction field	`final field, set once in constructor`	`init-only property, set once via initializer syntax`

DEFAULT TO AUTO-IMPLEMENTED PROPERTIES

`{ get; set; }` alone covers the overwhelming majority of cases — only expand to a full accessor body with explicit logic once a property genuinely needs validation, transformation, or a computed value.

A PROPERTY THAT LOOKS LIKE A FIELD CAN HIDE REAL, POSSIBLY EXPENSIVE LOGIC

Because a property's call site is indistinguishable from plain field access, a getter or setter with genuine work inside it — validation, a database call, anything non-trivial — is invisible at the call site in a way a Java method call never is. Keep property accessors cheap and side-effect-free as a matter of discipline, since nothing in the syntax itself signals otherwise.

Coding Challenges

CHALLENGE 1

Write a class Rectangle with auto-implemented Width and Height properties, and a read-only Area property computed via an expression-bodied get that multiplies them.

View solution

CHALLENGE 2

Write a class Customer with a Name property whose setter trims whitespace and rejects an empty string by throwing an exception, then demonstrate assigning " Alice " and reading back the trimmed result.

View solution

CHALLENGE 3

Write a class Product with an init-only Sku property, construct an instance using object-initializer syntax, then attempt to reassign Sku afterward and show the resulting compile error.

View solution

CHAPTER 4 QUICK REFERENCE

- Properties are real language syntax ({ get; set; }) — Java's getter/setter is purely a naming convention, per java1-4
- A property call site (customer.Name) looks like field access but can run real logic underneath, unlike Java's visibly-a-method-call getName()
- init (C# 9) allows object-initializer syntax to set a property exactly once, at construction — the mechanism Course 2's records build on
- internal is C#'s own fourth access level — assembly-scoped, structurally different from java1-4's folder/package-scoped package-private
- **Next chapter:** inheritance and polymorphism — virtual/override required explicitly, the reverse of java1-5's virtual-by-default

CHAPTER 5 OF 8

Inheritance & Polymorphism

COURSE 1 · CH 5

Inheritance & Polymorphism

The exact reverse of java1-5's virtual-by-default dispatch — and a gotcha the compiler only warns about

This is the chapter where C# and Java's shared architecture stops predicting shared behavior. `java1-5` made every instance method virtual unless told otherwise. C# does the opposite.

base and Inheritance Basics

```
class Animal {
    protected string Name;
    public Animal(string name) { Name = name; }
}

class Dog : Animal {                    // : instead of extends
    public Dog(string name) : base(name) {}  // base(...) instead of super(...)
}
```

Same underlying idea as `java1-5`, different keywords — single inheritance via `:`, a base constructor call via `base(...)`.

virtual and override — Required Explicitly

```
class Animal {
    public virtual string Speak() => "...";   // must opt IN to being overridable
}
class Dog : Animal {
    public override string Speak() => "Woof";  // must opt IN to overriding it
}

Animal a = new Dog();
a.Speak();  // "Woof" — dynamic dispatch, because BOTH keywords were present
```

Here's the real reversal: in C#, a method is **not** dynamically dispatched by default. `java1-5` established that every Java instance method is virtual unless explicitly marked `final` / `private` / `static`. C# requires the base class to mark a method `virtual` *and* the derived class to mark its own version `override` — without both keywords present, calling that method through a base-typed reference always runs the base class's own version, regardless of the object's real runtime type.

What Happens Without virtual/override — Method Hiding

```
class Animal {
    public string Speak() => "...";   // NOT virtual
}
class Dog : Animal {
    public new string Speak() => "Woof";  // hides, does not override
}

Animal a = new Dog();
a.Speak();          // "..." — Animal's own version, based on the DECLARED type
((Dog)a).Speak();   // "Woof" — only when accessed through the Dog-typed reference
```

Redeclaring a method with the same signature but no `override` creates an entirely different, non-polymorphic mechanism called **method hiding** — the `new` keyword makes this explicit. Which version runs depends on the reference's *declared* type, not the object's real runtime type — the opposite of what `java1-5` trained every reader to expect from an object hierarchy.

Why C# Requires Opt-In Virtual

Two real, deliberate reasons, not accidents of history: a non-virtual method call can be resolved at compile time, avoiding the runtime dispatch-table lookup a virtual call requires — a genuine performance difference at scale. And a base class author is forced to explicitly decide, and effectively document, which methods are meant to be extension points — rather than every method automatically becoming one whether the original author intended it or not, as `java1-5`'s virtual-by-default makes true for every Java class.

sealed — Preventing Further Overriding

```
public override sealed string Speak() => "Woof";  // no further class may override THIS override

public sealed class FinalBreed : Dog {}       // prevents inheritance from this class entirely
```

`sealed override` stops a further subclass from overriding an already-overridden method; `sealed class` prevents any further inheritance at all — genuinely comparable to Kotlin's own classes being closed to

inheritance by default, an ironic point of alignment given C# otherwise opts into virtual dispatch the same non-default way Kotlin does.

ASPECT	JAVA (JAVA1-5)	C#
Dispatch default	`virtual by default`	`non-virtual by default`
To enable dynamic dispatch	`nothing needed — always on`	`virtual (base) + override (derived), both required`
Redeclaring without opting in	`n/a — always overrides`	`method hiding — a different, non-polymorphic mechanism`
Compile-time feedback if forgotten	`n/a`	`a warning (CS0114), not an error`

MARK VIRTUAL ONLY WHEN A METHOD IS GENUINELY MEANT TO BE EXTENDED

C#'s opt-in model exists specifically so a base class's author states real intent — reach for virtual deliberately, as a design decision, not as a default habit carried over from Java.

FORGETTING OVERRIDE IS A WARNING, NOT AN ERROR — THE CODE STILL RUNS

Redeclaring a base method's signature without `override` or `new` compiles successfully with only a CS0114 warning ("hides inherited member") — the program builds and runs, silently producing method-hiding behavior instead of the polymorphism the programmer likely intended. This is genuinely easy to miss if warnings aren't being read.

Coding Challenges

CHALLENGE 1

Write a Shape base class with a virtual Area() method returning 0, and a Circle subclass that overrides it correctly. Store the Circle in a Shape-typed variable and call Area(), showing Circle's version runs.

View solution

CHALLENGE 2

Repeat Challenge 1, but make Shape's Area() NOT virtual, and have Circle redeclare it with new instead of override. Call Area() through a Shape-typed variable and through a Circle-typed variable, showing the two different results.

View solution

CHALLENGE 3

Write Challenge 2's Circle.Area() with neither override nor new (just a plain redeclaration), compile it, and report the exact CS0114 warning text produced, explaining why the code still compiles and runs despite it.

View solution

CHAPTER 5 QUICK REFERENCE

- C# requires virtual (base) AND override (derived) for dynamic dispatch — the exact reverse of java1-5's virtual-by-default
- Redeclaring without override creates method hiding (new keyword) — dispatch depends on the reference's declared type, not the object's real type
- Opt-in virtual buys compile-time-resolvable calls by default and forces explicit intent about what's an extension point
- sealed override stops further overriding; sealed class stops further inheritance entirely, echoing Kotlin's own closed-by-default classes
- Forgetting override produces only a CS0114 warning, not a compile error — the silent method-hiding gotcha still compiles and runs
- **Next chapter:** interfaces and abstract classes — default interface methods since C# 8, and explicit interface implementation

CHAPTER 6 OF 8

Interfaces & Abstract Classes

COURSE 1 · CH 6

Interfaces & Abstract Classes

A feature with no Java equivalent at all — resolving a name collision without picking a winner

This time Java arrived first: `java1-6`'s default interface methods shipped in Java 8 (2014), years before C# 8 (2019) added the same idea to C#. But C# didn't stop there — it added something Java's interface system has no equivalent for at all.

Interfaces — Recap

```
interface IMovable {
    void Move(double distance);  // implicitly public — no modifier written
}
```

Same underlying idea as `java1-6` — a pure contract, a class `implements` ... except C# spells that keyword `:`, the same syntax used for class inheritance. Interface members are implicitly `public`, with no modifier needed.

Default Interface Methods (C# 8) — Java Arrived First This Time

```
interface IMovable {
    void Move(double distance);
    string Stop() => "Stopping.";  // a real default body, C# 8 (2019)
}
```

C# 8 added default interface method bodies — the identical idea `java1-6` described for Java 8, five years earlier. Worth stating plainly, since it varies chapter to chapter: sometimes C# arrives first (switch expressions, records), and sometimes Java does, as here.

Multiple Interface Implementation — The Diamond Problem, Again

```
interface Greeter { string Greet() => "Hello from Greeter"; }
interface Welcomer { string Greet() => "Hello from Welcomer"; }

class Host : Greeter, Welcomer {
    public string Greet() => ((Greeter)this).Greet();  // explicit resolution required, like java1-6
}
```

The same narrow diamond problem `java1-6` covered resurfaces here — two conflicting default implementations force an explicit resolution, or the class fails to compile. The syntax differs (a cast to the interface type, rather than Java's `InterfaceName.super.method()`), but the underlying requirement is identical: the compiler refuses to guess.

Explicit Interface Implementation — No Java Equivalent

```
interface IPrintable { void Process(); }
interface ISavable { void Process(); }

class Document : IPrintable, ISavable {
    void IPrintable.Process() { Console.WriteLine("Printing..."); }  // TWO separate implementations —
    void ISavable.Process() { Console.WriteLine("Saving..."); }    // no collision at all
}

Document doc = new Document();
((IPrintable)doc).Process();  // "Printing..."
((ISavable)doc).Process();    // "Saving..." — the SAME object, two distinct behaviors
```

Here's the genuine reveal: `java1-6`'s own diamond-problem fix forces a single, explicit resolution — one implementation wins. C#'s **explicit interface implementation** lets a class keep *both* conflicting implementations simultaneously, each one only reachable through the specific interface type it belongs to. This resolves a real name-collision scenario Java's interface system has no mechanism for at all — Java would require renaming one of the two methods; C# doesn't.

Abstract Classes — Quick Recap

```
abstract class Shape {
    public abstract double Area();  // no body — subclasses must supply one
    public string Describe() => $"Area: {Area()}";
}
```

Same idea as `java1-6`'s own abstract classes — real state, constructors, and concrete methods alongside abstract ones, still limited to single inheritance.

FEATURE	JAVA (JAVA1-6)	C#
Default method bodies	`Java 8 (2014) — first`	`C# 8 (2019)`
Conflicting defaults from two interfaces	`forced single explicit override`	`forced single explicit resolution, same idea`
Same method name, two distinct behaviors	`not possible — must rename one`	`explicit interface implementation — both kept`

REACH FOR EXPLICIT INTERFACE IMPLEMENTATION FOR GENUINE NAME COLLISIONS

When two interfaces a class implements both need a method with the identical name but different meanings, explicit interface implementation keeps both behaviors distinguishable by which interface reference is used to call them — no renaming required on either side.

AN EXPLICITLY-IMPLEMENTED MEMBER ISN'T REACHABLE THROUGH THE CLASS TYPE

`doc.Process()` does not compile at all if `Process()` was only implemented explicitly for `IPrintable`/`ISavable` — it's only reachable via a variable or cast of the specific interface type, never through the concrete class type directly. This is easy to be caught out by the first time it's encountered.

Coding Challenges

CHALLENGE 1

Write an interface IGreetable with an abstract Hello() method and a default Bye() method, then a class implementing IGreetable that only supplies Hello(), demonstrating Bye() is inherited automatically.

View solution

CHALLENGE 2

Write two interfaces IPrintable and ISavable that both declare a Process() method, and a Document class using explicit interface implementation to give each a genuinely different body. Call both through interface-typed casts of the same instance.

View solution

CHALLENGE 3

Using Challenge 2's Document class, attempt to call doc.Process() directly on a Document-typed variable (not cast to either interface). Show the resulting compile error and explain why it happens.

View solution

CHAPTER 6 QUICK REFERENCE

- Default interface methods shipped in C# 8 (2019) — years after java1-6's own Java 8 (2014) version
- Conflicting defaults from two interfaces still force explicit resolution, the same requirement java1-6 covered for Java
- Explicit interface implementation lets one class keep two genuinely different same-named methods — no Java equivalent exists
- An explicitly-implemented member is reachable ONLY through the specific interface type, never through the concrete class type
- Abstract classes work the same as java1-6's own — state, constructors, and concrete methods alongside abstract ones, still single inheritance
- **Next chapter:** exception handling — every exception is unchecked, a real departure from java1-7's checked/unchecked split

CHAPTER 7 OF 8

Exception Handling

COURSE 1 · CH 7

Exception Handling

Every C# exception is unchecked — a deliberate, documented rejection of java1-7's checked/unchecked split

`java1-7` positioned Java's checked exceptions as a real middle ground between C++'s unenforced exceptions and Rust's `Result<T, E>`. C#'s designers looked at that same middle ground and deliberately rejected it — this chapter explains why, with their own stated reasoning.

try/catch/finally — Same Shape

```
try {
    int result = 10 / 0;
} catch (DivideByZeroException e) {
    Console.WriteLine(e.Message);
} finally {
    Console.WriteLine("Always runs");
}
```

Structurally identical to `java1-7` — `try`, a typed `catch`, an unconditional `finally`. The real divergence isn't the syntax.

Every C# Exception Is Unchecked

C# has no checked-exception concept at all. Every exception, without exception, behaves the way `java1-7` described for `RuntimeException` — a method can throw anything, with zero compiler-enforced acknowledgment, ever. No `throws` clause exists in C#'s syntax to require in the first place.

Why C# Rejected Checked Exceptions

This was a deliberate design decision, not an oversight — Anders Hejlsberg (`csharp1-1`'s own origin-story architect) has spoken publicly about the specific reasoning. Two real, documented critiques of `java1-7`'s own model: checked exceptions don't version well — adding one new checked exception to a widely-used

method's signature breaks every existing caller that doesn't already handle it, a real practical cost in library evolution. And in practice, checked exceptions get "handled" via empty catch blocks or wrapped indiscriminately in a generic `RuntimeException` just to satisfy the compiler — arguably making real code worse, not safer, than leaving the choice to the programmer's judgment.

XML Doc Comments as the Real (Weaker) Substitute

```
/// <summary>Withdraws funds from the account.</summary>
/// <exception cref="InsufficientFundsException">Thrown when balance is too low.</exception>
public void Withdraw(decimal amount) { /* ... */ }
```

C# lets an API author *document* what a method might throw via an XML doc comment's `<exception>` tag, surfaced in IntelliSense — but nothing here is compiler-enforced at all. It's purely informational, a real, direct contrast against `java1-7`'s own compiler-enforced `throws` clause.

Exception Filters — A Genuinely Unique Addition

```
try {
    ProcessOrder();
} catch (HttpRequestException e) when (e.StatusCode == 503) {
    Retry();  // only fires when BOTH the type AND the condition match
} catch (HttpRequestException e) {
    LogAndFail(e);  // every other HttpRequestException falls here
}
```

`catch (...) when (condition)` is a real feature with no direct Java equivalent — a catch block only fires when its exception type *and* a runtime condition both hold, avoiding a nested `if` statement (and a re-throw of everything that doesn't match) inside a broader catch.

using — C#'s Answer to try-with-resources

```
using var reader = new StreamReader(path);  // disposed automatically at the END of the enclosing scope
Console.WriteLine(reader.ReadLine());
// no closing brace needed — this is C# 8's using DECLARATION, more concise than java1-7's try(...)
```

Any type implementing `IDisposable` works with `using`, calling `Dispose()` automatically — genuinely comparable to `java1-7`'s `try`-with-resources/`AutoCloseable`, and to `cpp1-5`'s RAII. C# 8's `using`

declaration form (shown above, no extra braces) is even more concise syntactically than Java's own block-scoped `try(...)` syntax.

ASPECT	JAVA (JAVA1-7)	C#
Checked exceptions	yes — compiler-enforced catch or throws	no such concept exists
Documenting what a method might throw	throws clause — compiler-enforced	XML doc <exception> tag — informational only
Conditional catch	not built in — needs a nested if	catch (...) when (condition) — built in
Automatic resource cleanup	try-with-resources	using — including a more concise declaration form

REACH FOR EXCEPTION FILTERS INSTEAD OF RE-THROWING INSIDE A BROAD CATCH

`catch (...) when (condition)` keeps a narrow, conditional handling case cleanly separate from the general case, without needing an inner `if` and a manual re-throw to fall through to a second catch block.

NOTHING FORCES YOU TO CONSIDER A REALISTIC FAILURE MODE

Because C# has no checked-exception mechanism, the compiler never prompts a caller to think about what a method might throw — `java1-7`'s own compiler-enforced acknowledgment simply has no C# counterpart. Knowing what a method can throw relies entirely on documentation, testing, and discipline, not language enforcement.

Coding Challenges

CHALLENGE 1

Write a method that divides by zero inside a try block, catch DivideByZeroException specifically, and use a finally block that always runs. Then write a second method that throws a custom exception with no throws-style declaration anywhere, showing it compiles without any compiler warning at all.

View solution

CHALLENGE 2

Write a method that throws a custom exception with an int ErrorCode property, then use two catch blocks: one with a when filter that only fires for a specific error code, and a second, unfiltered catch for every other case.

View solution

CHALLENGE 3

Write a class implementing IDisposable with a Dispose() method that prints a message, then use it with a using declaration (not a using block) alongside code that throws an exception afterward in the same scope, showing Dispose() still runs.

View solution

CHAPTER 7 QUICK REFERENCE

- C# has no checked-exception concept — every exception behaves like java1-7's own RuntimeException, with zero compiler-enforced acknowledgment
- This was a deliberate, documented design choice (Anders Hejlsberg) — checked exceptions version poorly and get swallowed via empty catches/blanket rethrows in practice
- XML doc <exception> tags document possible throws for IntelliSense, but enforce nothing at compile time, unlike Java's throws clause
- catch (...) when (condition) — exception filters with no direct Java equivalent
- using (including C# 8's more concise using declaration) is C#'s IDisposable-based answer to try-with-resources
- **Next chapter:** collections and a first taste of LINQ, previewing Course 2's own deep dive

CHAPTER 8 OF 8

Collections & a First Taste of LINQ

COURSE 1 · CH 8

Collections & a First Taste of LINQ

C# had this idea in 2007 — seven years before java2-4's own Streams API

Fundamentals closes with C#'s everyday collections — and a first look at the feature that touches nearly every one of them, deferred to Course 2's own full chapter, but too central to leave out entirely here.

List<T> — A Growable Collection

```
List<string> names = new();
names.Add("Alice");
names.Add("Bob");
```

Nearly identical in behavior to `java1-8`'s own `List` / `ArrayList` — grows automatically, generic. C#'s generics are reified rather than erased, a real difference Course 2's own Generics In Depth chapter covers in full; this chapter stays at the surface level.

Dictionary<TKey, TValue>

```
Dictionary<string, int> ages = new();
ages["Alice"] = 30;   // indexer syntax — no .put() call needed
Console.WriteLine(ages["Alice"]);
```

Comparable to `java1-8` / `java2-2`'s own `Map` / `HashMap` — key-value pairs, average constant-time lookup. C#'s indexer syntax (`ages["Alice"]`) reads slightly more concisely than Java's explicit `.put()` / `.get()` calls, though the underlying idea is identical.

Iterating — foreach

```
foreach (var name in names) {
    Console.WriteLine(name);
}
```

Directly comparable to Java's own enhanced `for` loop — same purpose, different keyword.

A First Taste of LINQ

```
int[] numbers = { 1, 2, 3, 4, 5, 6 };

// method syntax
var evenSquares = numbers.Where(n => n % 2 == 0).Select(n => n * n);

// query syntax — SQL-like, built directly into the language grammar
var evenSquares2 = from n in numbers
                   where n % 2 == 0
                   select n * n;
```

LINQ — Language Integrated Query — shipped in C# 3.0, back in 2007, genuinely comparable to `java2-4`'s own Streams API but arriving **seven years earlier**. Another "C# first" moment, like `csharp1-3`'s own switch expressions. LINQ offers *two* syntaxes for the same underlying query: fluent method chaining (`.Where().Select()`), and a genuinely unique SQL-like **query syntax** baked directly into C#'s own grammar — something neither Java nor most other languages on this site offer as native syntax at all.

Why Preview LINQ Now

LINQ works over anything implementing `IEnumerable<T>` — which includes `List<T>`, `Dictionary<TKey,TValue>`, arrays, and every collection this chapter just introduced. That's exactly why it belongs here rather than waiting entirely for Course 2 — it's the natural next step for everything just covered, not a separate, unrelated topic.

ASPECT	JAVA STREAMS (JAVA2-4)	C# LINQ
Arrival	`Java 8 (2014)`	`C# 3.0 (2007) — first`
Fluent chaining	`.filter().map().collect()`	`.Where().Select()`
SQL-like query syntax	`not available`	`from...where...select, built into the grammar`

METHOD SYNTAX COMPOSES; QUERY SYNTAX READS NATURALLY FOR SQL-LIKE FILTERING

Method syntax chains cleanly with everything else in C#; query syntax often reads more naturally for filtering/joining operations that already feel SQL-like. Course 2's own LINQ In Depth chapter covers when to reach for each.

THIS IS ONLY A FIRST TASTE — THE REAL DEPTH IS COURSE 2'S

Deferred execution, the difference between `IEnumerable<T>` and `IQueryable<T>`, and LINQ's full operator vocabulary are all Course 2 material — this chapter only establishes that LINQ exists and roughly what it looks like.

Coding Challenges

CHALLENGE 1

Create a List<int> of at least six numbers, then use LINQ method syntax to filter for numbers greater than 10 and print the result using foreach.

View solution

CHALLENGE 2

Create a Dictionary<string, int> of at least four name/age pairs, then write the same query twice — once using LINQ method syntax and once using query syntax — to find everyone with age 30 or older, printing both results to show they match.

View solution

CHALLENGE 3

Write a short comment explaining why LINQ can operate over a List<T>, a Dictionary<TKey,TValue>, and a plain array all with the same syntax, referencing IEnumerable<T> directly.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 1 COMPLETE

- List<T> and Dictionary<TKey,TValue> behave comparably to java1-8/java2-2's List and Map
- foreach is directly comparable to Java's enhanced for loop
- LINQ shipped in C# 3.0 (2007), seven years before java2-4's own Streams API
- LINQ offers both fluent method syntax and a genuinely unique SQL-like query syntax built into the language grammar
- LINQ works over anything implementing IEnumerable<T> — lists, dictionaries, and arrays alike

- **C# Fundamentals is now complete.** Course 2 (Intermediate/Advanced) begins with Generics In Depth — C#'s reified generics, a genuine opposite design choice from java2-1's own type erasure.