

LUCAS EGA ACADEMY

C#



Curso Completo de C#

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

ÍNDICE

PRÓLOGO	5
Para quién es este libro	5
Por qué C#	6
Cómo está organizado este libro	7
Cómo estudiar este libro	8
Lo que necesitas	9
Una nota sobre los errores	9
UNIDAD 1: Introducción a C# y .NET	11
Explicación detallada	12
Ejemplos paso a paso	29
Práctica	34
Resumen de la unidad	40
UNIDAD 2: Variables, operadores y entrada de datos	42
Explicación detallada	43
Ejemplos paso a paso	60
Práctica	66
Resumen de la unidad	73
UNIDAD 3: Decisiones y ciclos while	75
Explicación detallada	76
Ejemplos paso a paso	98
Práctica	109
Resumen de la unidad	120
UNIDAD 4: Ciclos for/foreach y métodos	122
Explicación detallada	123
Ejemplos paso a paso	146
Práctica	158
Resumen de la unidad	170
UNIDAD 5: Arreglos y POO básica	172
Explicación detallada	173
Ejemplos paso a paso	184

Práctica	195
Resumen de la unidad	207
UNIDAD 6: Encapsulamiento, herencia y polimorfismo	210
Explicación detallada	211
Ejemplos paso a paso	221
Práctica	233
Resumen de la unidad	251
UNIDAD 7: Interfaces, colecciones y LINQ	253
Explicación detallada	254
Ejemplos paso a paso	274
Práctica	287
Resumen de la unidad	296
UNIDAD 8: Excepciones, archivos y cadenas	299
Explicación detallada	300
Ejemplos paso a paso	323
Práctica	335
Resumen de la unidad	349
UNIDAD 9: Asincronía, delegados y genéricos	351
Explicación detallada	353
Ejemplos paso a paso	381
Práctica	388
Resumen de la unidad	401
UNIDAD 10: Proyectos finales	403
Explicación detallada	404
Ejemplos paso a paso	410
Práctica	430
Resumen de la unidad	450
EXAMEN FINAL	452
Preguntas	452
Respuestas y soluciones	460

PRÓLOGO

Bienvenido a "Curso Completo de C#", el libro oficial de Lucas EGA Academy (lucasegaacademy.com) para aprender desde cero el lenguaje con el que se construyen aplicaciones de escritorio, videojuegos, aplicaciones móviles, servicios web y sistemas empresariales en todo el mundo. Este libro nace del curso gratuito de 30 clases de la academia, pero aquí tienes todo explicado con muchísimo más detalle: conceptos a fondo, analogías de la vida real, los errores típicos de principiantes, buenas prácticas de la industria, ejemplos paso a paso para probar en tu computadora, prácticas con soluciones explicadas y un examen final para medir lo que aprendiste.

C# (se pronuncia "si sharp") es un lenguaje moderno, elegante y poderoso, creado por Microsoft y hoy completamente abierto y multiplataforma gracias a .NET. Con lo que aprendas en este libro vas a poder escribir programas de consola, trabajar con datos, construir clases y sistemas orientados a objetos, manejar archivos, programar de forma asíncrona y terminar con proyectos completos que puedes presumir en tu portafolio. Y lo mejor: C# es uno de los lenguajes mejor pagados y con más oferta de empleo en el mercado hispanohablante.

Si este es tu primer lenguaje de programación, elegiste bien: C# te enseña a pensar con orden, te obliga a entender los tipos de datos y te da una base sólida para aprender después cualquier otro lenguaje como Java, Python o JavaScript. Si ya programas en otro lenguaje, C# te va a resultar familiar en muchas cosas, pero te va a enseñar disciplina: tipado fuerte, orientación a objetos bien hecha y herramientas profesionales que elevan tu nivel.

Para quién es este libro

Este libro es para ti si empiezas desde cero y quieres un lenguaje serio, con futuro laboral real y una comunidad enorme. Es para ti si estudias ingeniería, sistemas o computación y necesitas dominar C# para tus materias o tu servicio social. Es para ti si ya programas en Python, Java o JavaScript y quieres un lenguaje compilado, rápido y con tipado fuerte que te abra las puertas del desarrollo empresarial y de videojuegos.

También es para ti si sueñas con crear videojuegos (Unity usa C# como su lenguaje principal), si quieres trabajar en empresas que usan tecnología Microsoft, si te interesa el desarrollo backend con ASP.NET, o si simplemente quieres aprender a programar "bien", con las bases que distinguen a un aficionado de un profesional. No necesitas matemáticas avanzadas ni experiencia previa: solo una computadora, curiosidad y ganas de practicar todos los días.

Si vienes de otros cursos de Lucas EGA Academy, este libro encaja perfecto: ya dominas la lógica de programar, y ahora la vas a llevar a un lenguaje compilado, con tipos estrictos y orientación a objetos de verdad. Todo lo que aprendiste de HTML, CSS o JavaScript te sirve como base mental; aquí lo formalizamos.

Y si este es tu primer libro de la academia, no te preocupes: empezamos desde qué es un programa y cómo se ejecuta, y no damos nada por sentado. Eso sí, te voy a pedir una cosa desde el inicio: que programes todos los días, aunque sea media hora. La constancia le gana al talento, siempre.

Por qué C#

Hay muchos lenguajes para empezar a programar, así que vale la pena explicar por qué este libro apuesta por C#. La respuesta corta: porque es un lenguaje que te enseña bien, te emplea bien y te acompaña lejos.

Primero, C# te enseña a programar bien. Es un lenguaje de tipado fuerte: cada variable tiene un tipo claro (número entero, texto, decimal, verdadero/falso) y el compilador te avisa cuando intentas mezclar cosas que no van juntas. Al principio eso puede sentirse estricto, como un maestro exigente, pero es justo lo que un principiante necesita: en lugar de que tu programa falle misteriosamente a la mitad, el compilador te marca el error antes de que corra el programa. Aprendes a pensar con precisión, y esa precisión te sirve en todos los lenguajes.

Segundo, C# te emplea bien. Es el lenguaje estrella del ecosistema .NET de Microsoft: se usa en bancos, aseguradoras, gobiernos y empresas de todos los tamaños para sistemas internos, aplicaciones de escritorio, servicios web con ASP.NET y aplicaciones móviles. Además es el lenguaje de Unity, el motor con el que se hacen miles de videojuegos independientes y profesionales. Saber C# abre puertas reales, con sueldos reales, en tu país y en remoto para empresas extranjeras.

Tercero, C# te acompaña lejos. No es un lenguaje "de juguete" para aprender y luego abandonar: es el mismo lenguaje que usan profesionales con diez o quince años de experiencia. Lo que aprendas aquí (clases, herencia, interfaces, LINQ, programación asíncrona) son exactamente las herramientas que se usan en proyectos grandes. Y .NET hoy es gratuito, de código abierto y multiplataforma: el mismo código corre en Windows, Linux y macOS.

Hay otra razón, más personal: C# es un lenguaje bonito. Su sintaxis es limpia, sus mensajes de error son claros comparados con otros lenguajes compilados, y las herramientas (el SDK de .NET y Visual Studio Code) son gratuitas y excelentes. Programar en C# se siente

moderno, no como pelear con tecnología de hace cuarenta años. Vas a disfrutar el proceso, y eso importa: nadie aprende bien algo que odia.

Una advertencia honesta: C# exige más rigor que lenguajes como Python o JavaScript. Vas a declarar tipos, vas a pelear con el compilador y vas a escribir más líneas para lograr lo mismo. No te asustes: cada una de esas líneas te está enseñando algo. Al final del libro vas a mirar atrás y vas a entender por qué los programadores profesionales prefieren que la máquina les marque los errores temprano en lugar de descubrirlos tarde, cuando ya hay usuarios reales usando el programa.

Cómo está organizado este libro

El libro tiene 10 unidades que agrupan las 30 clases del curso en video de Lucas EGA Academy. Cada unidad cubre tres clases y sigue siempre la misma estructura para que sepas qué esperar: primero una introducción que te ubica (qué vas a aprender, por qué importa y dónde se usa en la vida real), después la explicación detallada con analogías, errores comunes de principiantes y buenas prácticas profesionales, luego ejemplos paso a paso con código completo que puedes correr en tu computadora, después una sección de práctica con ejercicios y sus soluciones explicadas, y al final un resumen con los puntos clave.

Estas son las 10 unidades:

- UNIDAD 1: Introducción a C# y .NET. Qué es C#, qué es .NET, cómo instalar el SDK, tu primer programa con Console.WriteLine y cómo compilar y ejecutar desde la terminal.
- UNIDAD 2: Variables, operadores y entrada de datos. Tipos de datos (int, double, bool, string, char), declaración de variables, operadores aritméticos, lógicos y de comparación, y cómo leer datos del teclado con Console.ReadLine.
- UNIDAD 3: Decisiones y ciclos while. Condicionales if, else if y else, switch, operadores lógicos combinados, y los ciclos while y do-while para repetir acciones.
- UNIDAD 4: Ciclos for y foreach, y métodos. El ciclo for clásico, el ciclo foreach para recorrer colecciones, y cómo crear tus propios métodos con parámetros y valores de retorno.
- UNIDAD 5: Arreglos y POO básica. Arreglos unidimensionales y multidimensionales, y tu primera clase: objetos, atributos, constructores y la palabra this.
- UNIDAD 6: Encapsulamiento, herencia y polimorfismo. Propiedades, modificadores de acceso, herencia con la palabra base, métodos virtuales y override, y polimorfismo en acción.

- UNIDAD 7: Interfaces, colecciones y LINQ. Interfaces, la lista genérica List<T>, diccionarios, y consultas elegantes con LINQ (Where, Select, OrderBy).
- UNIDAD 8: Excepciones, archivos y cadenas. Manejo de errores con try, catch y finally, lectura y escritura de archivos, y manipulación avanzada de texto con la clase string.
- UNIDAD 9: Asincronía, delegados y genéricos. Programación asíncrona con async y await, delegados y eventos, clases y métodos genéricos, y tipos anulables.
- UNIDAD 10: Proyectos finales. Tres proyectos completos que integran todo lo aprendido: un sistema de calificaciones, un inventario con archivos y un mini juego por turnos.

Al final del libro encontrarás el EXAMEN FINAL: 50 preguntas que cubren las 10 unidades (opción múltiple, verdadero o falso y preguntas abiertas de código), con sus respuestas y soluciones explicadas. Úsalo como tu prueba de graduación: si sacas 40 o más, dominas C# a nivel intermedio y estás listo para el siguiente paso, que puede ser ASP.NET, Unity o el curso que elijas en la academia.

Cómo estudiar este libro

No leas este libro como una novela. La programación se aprende con las manos en el teclado, así que te propongo este método, probado con miles de estudiantes de la academia:

1. Lee la explicación de cada unidad con calma. Si algo no queda claro a la primera, relee: los conceptos de orientación a objetos y de asincronía, por ejemplo, les cuestan a todos al inicio. Es normal, no eres tú, es el tema.
2. Teclea TÚ MISMO cada ejemplo de código. No copies y pegues: escribe el programa línea por línea, compilalo con "dotnet run" y observa qué hace. Tus dedos también tienen que aprender, y al teclear descubres detalles que leyendo se te escapan (un punto y coma, una mayúscula, el orden de los parámetros).
3. Rompe el código a propósito. Cambia valores, quita líneas, escribe mal un nombre de variable y lee el error que te muestra el compilador. En C# los errores del compilador son tus maestros: cada mensaje en rojo te está enseñando una regla del lenguaje. Aprende a leerlos sin miedo.
4. Haz los ejercicios de práctica sin mirar la solución primero. Si te atorras más de treinta minutos, mira la solución, entiéndela línea por línea, ciérrala y luego resuélvelo solo desde cero. El objetivo no es "pasar" el ejercicio, es que tu cerebro haga las conexiones solo.
5. No avances con dudas. C# es un lenguaje donde cada tema se construye sobre el anterior: si no entiendes bien los métodos, las clases te van a costar el doble; si no

entiendes las clases, la herencia te va a doler. Repasa la unidad las veces que necesites. Volver atrás no es retroceder, es cimentar.

6. Al terminar cada unidad, lee el resumen y pregúntate: ¿podría explicarle esto a alguien más con mis palabras? Si la respuesta es sí, vas bien. Si titubeas, repasa. Enseñar (aunque sea en voz alta para ti mismo) es la prueba definitiva de que entendiste.

Una hora diaria de práctica real vale más que siete horas un solo día a la semana. La programación es un hábito muscular y mental: la constancia le gana a las maratones.

Lo que necesitas

Para seguir este libro necesitas tres cosas, y dos son gratis:

1. El SDK de .NET (gratuito). Es el kit oficial de Microsoft con el compilador y las herramientas para crear, compilar y correr programas de C#. En la Unidad 1 te guío paso a paso para instalarlo en Windows, Linux y macOS, y para verificar que quedó bien instalado con el comando "dotnet --version".
2. Un editor de código. Te recomiendo Visual Studio Code (gratuito) con la extensión oficial de C#. Es ligero, potente y es el que usamos en todos los ejemplos del libro. Si prefieres, también puedes usar Visual Studio Community, que es gratis y trae todo integrado.
3. Ganas de practicar todos los días. Esta es la única que no se descarga. Sin ella, los otros dos no sirven de nada; con ella, no hay tema que se te resista.

Una nota sobre los errores

Vas a ver muchos mensajes de error en este camino: "no se encontró el tipo", "no se puede convertir implícitamente", "se esperaba ;", "NullReferenceException". Te lo digo desde ahora para que no te asustes: es normal, les pasa a todos, y me sigue pasando a mí después de años programando. Los errores no son fracasos: son el lenguaje con el que la computadora te dice qué necesita.

De hecho, una de las grandes ventajas de C# es que su compilador es estricto y te avisa de los problemas ANTES de que el programa corra. Cada error que corriges te deja una lección. Este libro dedica espacio en cada unidad a los errores más comunes del tema y cómo resolverlos, porque un programador no es alguien que no se equivoca: es alguien que sabe leer los errores y seguir adelante.

Échale ganas, ve a tu ritmo y no te saltes las prácticas. Al final de este libro no solo vas a "saber C#": vas a tener proyectos reales funcionando, hechos con tus propias manos, y la base sólida para entrar al desarrollo profesional, a los videojuegos con Unity o al backend con ASP.NET. Eso ya es ser programador.

Nos vemos en la Unidad 1. A programar.

Esteban Lucas EGA Academy - lucasegaacademy.com

UNIDAD 1: Introducción a C# y .NET

Imagina que quieres construir una casa. Podrías empezar cortando madera con un serrucho de mano, clavando tabla por tabla, y después de meses tendrías algo habitable. O podrías usar una constructora con maquinaria, planos hechos por arquitectos y materiales modernos: en semanas tendrías una casa sólida, con instalación eléctrica, plomería y acabados. Aprender C# es como elegir la segunda opción. En el mundo de la programación existen decenas de lenguajes, y muchos sirven para construir software, pero C# es uno de esos lenguajes que te da la maquinaria completa: un ecosistema maduro, herramientas profesionales gratuitas, una comunidad enorme y la capacidad de crear casi cualquier cosa que se te ocurra, desde videojuegos hasta aplicaciones de bancos, pasando por páginas web, aplicaciones de escritorio y apps para teléfonos.

Esta primera unidad es tu puerta de entrada. Aquí no vamos a dar nada por sentado: vamos a explicar qué es C#, de dónde viene, por qué Microsoft lo creó y por qué hoy es uno de los lenguajes más demandados en el mercado laboral. Vamos a comparar C# con otros lenguajes populares para que entiendas dónde brilla y cuándo conviene usarlo. Vas a conocer el ecosistema .NET, que es el conjunto de herramientas, bibliotecas y el entorno de ejecución que hacen que tu código C# funcione en Windows, en macOS, en Linux, en la nube y hasta en consolas de videojuegos. Y lo más importante: al terminar esta unidad vas a tener tu computadora configurada como la de un programador profesional y vas a correr tu primer programa en C#. Ese momento, cuando ves tus primeras palabras aparecer en la pantalla porque TÚ escribiste el código que las produjo, es el momento en que dejas de ser un espectador y te conviertes en un creador de software. No lo subestimes: todos los programadores profesionales del mundo pasaron por ese instante, y es el primero de miles.

Lo que vas a aprender en esta unidad se divide en tres grandes bloques. Primero, la teoría fundamental: qué es C#, su historia (incluyendo la figura de Anders Hejlsberg, el genio danés que lo diseñó), qué es .NET y cómo se relacionan, y cómo funciona por dentro la compilación de tu código, incluyendo el concepto de compilación JIT (Just-In-Time) que hace que C# sea rápido sin que tú tengas que hacer nada especial. Segundo, la parte práctica de instalación: vas a instalar el .NET SDK (el kit de desarrollo, gratis y oficial), el editor Visual Studio Code (ligero, gratis y potentísimo), y vas a aprender a usar la terminal con los comandos "dotnet new console" y "dotnet run", que son la forma moderna y profesional de crear y ejecutar proyectos. Tercero, tu primer programa: vamos a desarmarlo pieza por pieza para que entiendas qué hace cada línea, qué es el método Main, qué significa "using", qué son los namespaces, cómo funciona Console.WriteLine y cómo se escriben comentarios. Nada de copiar código sin entender: aquí cada línea tiene una explicación.

¿Por qué importa todo esto en la vida real? Porque C# no es un lenguaje de juguete para ejercicios de escuela. Es el lenguaje con el que se construyen sistemas bancarios reales, el backend de miles de empresas, una enorme cantidad de videojuegos hechos con Unity (el motor de videojuegos más usado del mundo para juegos independientes y móviles), aplicaciones de escritorio para Windows, y servicios en la nube que atienden a millones de usuarios. Cuando termines esta unidad tendrás la base para todo lo demás: cada unidad posterior de este libro asume que ya sabes crear un proyecto, escribir código y ejecutarlo. Es como aprender a encender el coche y a usar el volante antes de salir a la carretera: sin esto, nada más avanza.

Un consejo de colega antes de empezar: programa de verdad, no solo leas. Cada vez que veas un bloque de código en este libro, escríbelo tú mismo en tu computadora y ejecútalo. Los errores que cometes al escribir (un punto y coma olvidado, una mayúscula mal puesta) son tus mejores maestros. C# es un lenguaje estricto con los detalles, y esa exigencia, que al principio puede frustrarte, es exactamente lo que te va a convertir en un programador cuidadoso y profesional. Vamos a ello.

Explicación detallada

Qué es C# y qué es .NET (y por qué siempre van juntos)

C# (se pronuncia "si sharp", como la nota musical do sostenido) es un lenguaje de programación moderno, orientado a objetos y de tipado fuerte, creado por Microsoft. Pero esa definición de diccionario no te dice mucho si vas empezando, así que vamos a desmenuzarla con calma.

Un lenguaje de programación es, en esencia, una forma de darle instrucciones a la computadora usando palabras y símbolos que los humanos podemos leer y escribir. La computadora, en el fondo, solo entiende unos y ceros (código binario). Entre tus palabras y esos unos y ceros hay un traductor: el compilador y el entorno de ejecución. C# es el idioma que tú escribes; .NET es todo el sistema que lo traduce, lo ejecuta y le da superpoderes.

Piensa en C# como el español que hablas y en .NET como el país donde vives: el idioma te sirve para comunicarte, pero el país te da las carreteras, los hospitales, las escuelas y los servicios que hacen que tu vida funcione. De la misma forma, .NET te da miles de bibliotecas ya hechas (código que otros ya escribieron y probaron para que tú no reinventes la rueda), herramientas para publicar tus programas, y un entorno de ejecución que hace que tu programa corra igual en Windows, en Linux o en macOS.

.NET, en su forma moderna (.NET 5, 6, 7, 8 y versiones posteriores), es gratuito, de código abierto y multiplataforma. Esto último es importante: multiplataforma significa que el mismo código que escribes en tu computadora con Windows puede ejecutarse en un servidor Linux en la nube o en una Mac, sin cambios. Hace años esto no era así (el viejo ".NET Framework" solo funcionaba en Windows), pero Microsoft cambió de estrategia y hoy .NET compete de tú a tú con ecosistemas como Java o Node.js en cualquier sistema operativo.

Entonces, la relación es así de simple: C# es el lenguaje (lo que escribes), .NET es la plataforma (lo que lo ejecuta y le da bibliotecas). Casi siempre que alguien dice "programo en C#", está programando en C# sobre .NET. En este libro, cuando digamos ".NET 8" o superior, nos referimos a la versión moderna y multiplataforma que vas a instalar.

Historia de C#: Anders Hejlsberg y el año 2000

Toda herramienta poderosa tiene una historia, y la de C# vale la pena porque explica muchas de sus decisiones de diseño. A finales de los años 90, Microsoft tenía un problema: su lenguaje estrella para Windows era Visual Basic, fácil de usar pero limitado, y también promovía C++, potentísimo pero difícil y peligroso (un error de memoria en C++ puede tumbar todo el sistema). Mientras tanto, Sun Microsystems había lanzado Java en 1995, un lenguaje moderno, orientado a objetos, con recolección de basura (memoria automática) y la promesa de "escribe una vez, ejecuta en cualquier parte". Java estaba ganando terreno rapidísimo, sobre todo en empresas.

Microsoft decidió crear su propia respuesta: un lenguaje moderno que combinara la productividad de Visual Basic con la potencia de C++, pero diseñado desde cero para su nueva plataforma .NET. Para liderar el diseño contrataron a Anders Hejlsberg, un ingeniero de software danés que ya era una leyenda: había creado Turbo Pascal en los años 80 (el entorno que enseñó a programar a toda una generación) y había sido el arquitecto principal de Delphi. Hejlsberg no era un teórico de escritorio: era alguien que entendía lo que los programadores reales necesitaban día a día.

C# se anunció oficialmente en el año 2000, junto con la plataforma .NET. El nombre "C#" es un guiño: en música, el símbolo # significa "sostenido", un semitono arriba. C# es, simbólicamente, "C un paso arriba": toma la sintaxis familiar de C, C++ y Java (las llaves, los puntos y comas, los paréntesis) y la eleva con ideas modernas. Si alguna vez ves código Java y código C#, notarás que se parecen mucho: no es casualidad, ambos beben de la misma tradición.

Desde el 2000, C# no ha dejado de evolucionar. Cada pocos años sale una versión nueva con mejoras: C# 2.0 trajo los genéricos, C# 3.0 trajo LINQ (una forma elegante de consultar datos),

versiones posteriores agregaron programación asíncrona (async/await, fundamental para aplicaciones modernas), y las versiones recientes (C# 9, 10, 11, 12) han simplificado la sintaxis para que escribir código sea más rápido y legible, como los "top-level statements" que verás en este libro (programas que empiezan directamente, sin ceremonias). Hoy C# es un lenguaje maduro con más de dos décadas de mejoras continuas, respaldado por una de las empresas de tecnología más grandes del mundo y por una comunidad open source muy activa. No es una moda pasajera: es una apuesta segura para tu carrera.

Por qué aprender C#: tipado fuerte, multiplataforma y campo laboral

¿Por qué elegir C# entre tantos lenguajes? Vamos a verlo por partes, comparando con otros lenguajes populares para que tengas el mapa completo.

Primero, el tipado fuerte. En C#, cada variable tiene un tipo de dato declarado (número entero, texto, decimal, etc.) y el compilador verifica que no mezcles peras con manzanas. Si intentas guardar un texto en una variable numérica, el compilador te lo impide ANTES de ejecutar el programa. Esto contrasta con lenguajes de tipado dinámico como Python o JavaScript, donde ese error solo aparece cuando el programa ya está corriendo (y a veces, en producción, frente a usuarios reales). El tipado fuerte es como tener un revisor que checa tu tarea antes de entregarla: al principio sientes que te regaña mucho, pero con el tiempo agradeces que atrape errores temprano. Para proyectos grandes y equipos profesionales, esto es oro puro: menos errores en producción, código más fácil de mantener y herramientas (como el autocompletado del editor) que te ayudan muchísimo porque "saben" qué tipo es cada cosa.

Segundo, multiplataforma de verdad. Con .NET moderno, tu programa de consola corre en Windows, Linux y macOS. Con ASP.NET Core (el framework web de .NET) construyes el backend de sitios y APIs que se hospedan en cualquier nube. Con MAUI puedes crear aplicaciones móviles y de escritorio. Y con Unity, el motor de videojuegos, programas la lógica de juegos que se publican en PC, consolas, celulares y hasta realidad virtual. Un solo lenguaje, muchísimos destinos. Compáralo con Swift (casi solo Apple) u Objective-C: C# te abre más puertas con una sola inversión de aprendizaje.

Tercero, los campos donde C# brilla en la vida real:

- Videojuegos con Unity: Unity es el motor más usado para juegos independientes, juegos móviles y prototipos de estudios grandes. Todo el scripting de Unity se hace en C#. Si sueñas con hacer videojuegos, C# es prácticamente obligatorio.
- Aplicaciones empresariales: bancos, aseguradoras, gobiernos y grandes empresas usan .NET para sus sistemas internos y de cara al cliente. Son empleos estables y bien pagados.

- Backend con ASP.NET Core: para construir APIs y servicios web rápidos y escalables. Muchas empresas lo eligen por su rendimiento (en benchmarks, ASP.NET Core compite con los frameworks más rápidos del mundo).
- Aplicaciones de escritorio Windows: con WPF, WinForms o MAUI.
- Nube: Microsoft Azure está optimizado para .NET, y la integración es natural.

Comparación rápida con otros lenguajes: Python es más fácil para empezar y domina en ciencia de datos e inteligencia artificial, pero es más lento y su tipado dinámico complica proyectos gigantes. JavaScript es el rey del navegador (no hay alternativa ahí), pero fuera del navegador compite con Node.js y su ecosistema es más caótico. Java es el rival más directo de C# (similar filosofía, similar mercado empresarial); elegir entre ellos muchas veces depende de la región y la empresa, pero C# suele ofrecer herramientas más modernas y un lenguaje que evoluciona más rápido. C++ es más potente en control de hardware y rendimiento extremo (motores gráficos, sistemas embebidos), pero mucho más difícil y peligroso para un principiante. En resumen: C# está en el punto dulce entre potencia, productividad, seguridad y empleabilidad.

Cómo se ejecuta tu código: compilación y JIT

Aquí viene una de las partes más interesantes: ¿qué pasa realmente cuando escribes código C# y lo ejecutas? Entender esto te va a dar una intuición que te servirá toda tu carrera.

Cuando escribes un programa en C#, lo escribes en texto plano (archivos .cs). La computadora no puede ejecutar ese texto directamente. Entonces entra el compilador de C# (llamado Roslyn): su trabajo es traducir tu código a un lenguaje intermedio llamado IL (Intermediate Language, antes MSIL) y empaquetarlo en un ensamblado (archivos .dll o .exe). Este IL no es código máquina todavía: es una especie de "esperanto" de .NET, un código intermedio independiente del sistema operativo y del procesador.

Luego, cuando ejecutas tu programa, entra en escena el CLR (Common Language Runtime), el entorno de ejecución de .NET. El CLR toma ese IL y, justo en el momento de la ejecución, lo traduce a código máquina nativo del procesador donde está corriendo. A esta traducción de último momento se le llama compilación JIT (Just-In-Time, "justo a tiempo"). ¿Por qué hacerlo así en dos pasos en lugar de compilar directo a código máquina? Por tres razones poderosas:

1. Portabilidad: el mismo ensamblado .dll puede ejecutarse en Windows, Linux o macOS, porque la traducción final a código máquina la hace el CLR de cada plataforma. Tú compilas una vez y corre en todos lados.

2. Optimización en contexto: como el JIT compila mientras el programa corre, puede optimizar basándose en cómo se está usando realmente el código en esa máquina específica (qué métodos se llaman más, qué procesador hay, etc.). Es como un chef que ajusta la receta según los ingredientes frescos del día.
3. Servicios del CLR: mientras tu programa corre, el CLR le da servicios gratis: recolección de basura (libera automáticamente la memoria que ya no usas, adiós a una familia entera de errores de C++), verificación de tipos, manejo de excepciones y seguridad.

Una analogía de la vida real: imagina que escribes una receta en español neutro (tu código C#), un traductor la convierte a un idioma intermedio universal (IL), y cuando alguien va a cocinarla en su país, un chef local la adapta al momento a los ingredientes y utensilios de su cocina (JIT). El resultado: la misma receta funciona en México, en Japón y en Alemania, y además cada chef la optimiza para su cocina.

En la práctica, hay dos comandos que vas a usar todo el tiempo: "dotnet build" compila tu código (traduce C# a IL y verifica que no haya errores), y "dotnet run" compila (si hace falta) y ejecuta tu programa en un solo paso. Durante el desarrollo usarás "dotnet run" casi siempre; "dotnet build" lo usarás cuando quieras verificar que todo compila sin ejecutar, por ejemplo antes de compartir tu código.

Un error común de principiantes: pensar que si el programa "compiló" entonces "está bien". Compilar solo significa que la sintaxis es correcta y los tipos cuadran; tu programa puede compilar perfectamente y aun así hacer algo incorrecto (un error de lógica). El compilador es tu revisor de ortografía y gramática, no tu maestro de redacción: te dice si escribiste bien las palabras, no si la historia tiene sentido. Esa distinción te va a acompañar siempre: errores de compilación (el programa ni arranca) versus errores de lógica o de ejecución (arranca pero hace algo mal o se cae a la mitad).

Instalación del .NET SDK: tu kit de herramientas

Para programar en C# necesitas instalar el .NET SDK (Software Development Kit, kit de desarrollo de software). Ojo con la terminología, porque confunde a muchos principiantes: el SDK es lo que necesitas para DESARROLLAR (incluye el compilador, las herramientas de línea de comandos y las bibliotecas); el "runtime" solo sirve para EJECUTAR programas ya hechos. Como tú vas a crear programas, instala el SDK, que de todos modos incluye el runtime.

El SDK es gratuito y se descarga desde el sitio oficial de Microsoft (dotnet.microsoft.com). Busca la versión LTS (Long Term Support, soporte a largo plazo) más reciente; LTS significa que Microsoft le dará actualizaciones y soporte por varios años, ideal para aprender y para

proyectos serios. En este libro usamos .NET 8 o superior, que es donde los "top-level statements" (programas simplificados que verás enseguida) funcionan de maravilla.

Pasos generales de instalación (los detalles exactos cambian un poco según tu sistema, pero la idea es la misma):

1. Entra al sitio oficial de .NET y descarga el instalador del SDK para tu sistema operativo (Windows, macOS o Linux).
2. Ejecuta el instalador y sigue el asistente (en Windows es un .exe de siguiente-siguiente; en macOS un .pkg; en Linux suele ser por el gestor de paquetes o un script).
3. Abre una terminal (en Windows puedes usar PowerShell, CMD o la terminal integrada de VS Code; en macOS y Linux, la terminal del sistema) y escribe:

```
dotnet --version
```

Si la instalación salió bien, verás un número de versión como "8.0.100" o similar. Ese número confirma que el comando "dotnet" (la navaja suiza de .NET) está disponible en tu sistema. Si la terminal dice que no reconoce el comando, lo más común es que necesites cerrar y volver a abrir la terminal (para que recargue las rutas del sistema) o reiniciar la computadora. No te frustres: este es el problema número uno de instalación y tiene esa solución tan simple.

El comando "dotnet" es tu centro de mando. Con él creas proyectos, los compilas, los ejecutas, agregas bibliotecas y los publicas. En esta unidad usaremos tres de sus trucos: "dotnet new" (crear), "dotnet build" (compilar) y "dotnet run" (ejecutar). Más adelante en tu carrera usarás muchos más, pero estos tres son el 90 por ciento del trabajo diario.

Buena práctica profesional desde el día uno: mantén tu SDK actualizado dentro de la misma versión mayor. Las actualizaciones menores traen parches de seguridad y mejoras de rendimiento gratis. Y trabaja siempre con una versión LTS para proyectos que te importen, porque las versiones "STS" (soporte estándar, de vida corta) caducan rápido.

Visual Studio Code: tu editor de código

Puedes escribir código C# en el Bloc de notas, pero sería como construir una casa con un martillo de juguete. Los programadores usamos editores o IDEs que nos ayudan: colorean el código, detectan errores mientras escribes, autocompletan y depuran. Tienes dos grandes opciones gratuitas:

- Visual Studio Code (VS Code): un editor ligero, rapidísimo, gratuito y multiplataforma (Windows, macOS, Linux). Con la extensión de C# se convierte en un entorno muy capaz.

Es el que usaremos en este libro porque funciona igual en todos los sistemas y es el estándar de la industria para muchísimos lenguajes.

- Visual Studio (la versión "grande", solo Windows y Mac): un IDE completísimo y pesado, excelente, pero excesivo para empezar y no disponible en Linux.

No los confundas por el nombre parecido: Visual Studio Code es el editor ligero; Visual Studio es el IDE gigante. Para este libro, instala Visual Studio Code desde su sitio oficial (code.visualstudio.com). Después, dentro de VS Code, abre la pestaña de extensiones (el icono de cuadritos en la barra lateral) y busca "C#", la extensión oficial de Microsoft (publicada por Microsoft). Instálala. También te recomiendo la extensión "C# Dev Kit" si quieres una experiencia aún más cercana a Visual Studio, aunque la extensión base de C# es suficiente para todo este libro.

Con VS Code y la extensión de C# obtienes gratis tres superpoderes:

1. Resaltado de sintaxis: el código se pinta de colores según su función (palabras clave, textos, números, comentarios), lo que hace muchísimo más fácil leerlo.
2. IntelliSense (autocompletado inteligente): cuando empiezas a escribir, el editor te sugiere métodos y te muestra su documentación. Es como tener un copiloto que conoce todas las bibliotecas de memoria.
3. Detección de errores en vivo: las líneas con problemas se subrayan en rojo antes de que ejecutes nada, y al pasar el mouse te explican qué está mal.

Error común de principiantes: instalar VS Code pero olvidar la extensión de C#. Sin ella, VS Code trata tu archivo .cs como texto plano: sin colores, sin ayuda, sin detección de errores. Si tu código se ve todo blanco y el editor no te ayuda, casi seguro te falta la extensión.

Otro error común: abrir en VS Code un archivo .cs suelto en lugar de la carpeta del proyecto. VS Code trabaja mejor cuando abres la CARPETA completa del proyecto (Archivo > Abrir carpeta), porque así entiende la estructura, encuentra las referencias y te da todo el IntelliSense. Acostúmbrate desde ahora: se abre la carpeta del proyecto, no archivos sueltos.

Tu primer proyecto: dotnet new console y dotnet run

Ya tienes el SDK y el editor. Ahora vas a crear tu primer proyecto como lo hacen los profesionales: con la terminal y el comando "dotnet". Un "proyecto" en .NET es una carpeta con una estructura estándar: contiene tu código, un archivo de configuración del proyecto (con extensión .csproj) y, cuando compilas, las carpetas de salida. Usar la estructura estándar es importante porque así cualquier programador .NET del mundo entiende tu proyecto al instante, y las herramientas funcionan sin configuración extra.

Vamos paso a paso. Abre una terminal y ejecuta estos comandos (después del símbolo \$ va lo que TÚ escribes):

```
$ dotnet new console -n MiPrimerPrograma
```

¿Qué hace este comando? "dotnet new console" le dice a .NET: créame un proyecto nuevo de tipo "console" (aplicación de consola, es decir, un programa que se comunica por texto en la terminal, sin ventanas gráficas). El parámetro "-n MiPrimerPrograma" le pone nombre a tu proyecto (y a su carpeta). Al ejecutarlo verás mensajes como "The template Console App was created successfully". Ya tienes tu primer proyecto.

Ahora entra a la carpeta y ejecútalo:

```
$ cd MiPrimerPrograma
$ dotnet run
```

"dotnet run" compila tu proyecto y lo ejecuta en un solo paso. Si todo salió bien, verás en la terminal:

```
Hello, World!
```

Felicidades: acabas de crear, compilar y ejecutar tu primer programa en C#. Ese "Hello, World!" (Hola, mundo) no es casualidad: es la tradición más antigua de la programación. Desde 1978, cuando apareció en el libro "El lenguaje de programación C" de Kernighan y Ritchie, el primer programa de todo programador imprime "Hello, World!". Es un rito de iniciación: si tu entorno puede mostrar esas palabras, todo lo demás (compilador, runtime, editor) funciona.

¿Qué hay dentro de la carpeta de tu proyecto? Si la exploras verás:

- MiPrimerPrograma.csproj: el archivo de proyecto. Es un XML sencillo que dice "esto es un proyecto de .NET 8, es un ejecutable, usa C# moderno". Normalmente no necesitas tocarlo al empezar, pero es bueno saber que existe.
- Program.cs: tu código fuente. Aquí está el "Hello, World!". Este es el archivo que vas a editar.

- obj/ y bin/: carpetas generadas automáticamente al compilar. En obj/ van archivos temporales de compilación; en bin/ va tu programa ya compilado, listo para ejecutarse. Buena práctica: nunca edites nada dentro de obj/ ni bin/ a mano; son territorio del compilador. Si algo raro pasa al compilar, borrar obj/ y bin/ y volver a compilar resuelve muchos misterios.

Vamos a abrir Program.cs en VS Code para ver qué generó la plantilla. Con la terminal dentro de la carpeta del proyecto, escribe "code ." (el punto significa "abre la carpeta actual en VS Code"). Verás algo así:

```
using System;

namespace MiPrimerPrograma;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

O, si tu plantilla es de las más nuevas, verás la versión simplificada con top-level statements:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

No te preocupes si ves una u otra: ambas hacen exactamente lo mismo. En la siguiente sección vamos a desarmar este programa línea por línea hasta que cada símbolo tenga sentido. Por ahora, lo importante es que ya sabes el flujo profesional completo: crear con "dotnet new console", entrar a la carpeta, y ejecutar con "dotnet run". Repítelo un par de veces hasta que te salga sin pensar, porque lo vas a usar en cada unidad de este libro.

Un consejo práctico: crea una carpeta general para tus prácticas, por ejemplo "csharp-practicas" en tus documentos, y dentro de ella crea un proyecto por ejercicio o por unidad. Mantener tus proyectos organizados desde el principio te ahorra el caos de tener 50 carpetas regadas por el escritorio con nombres como "prueba1", "prueba1-final", "prueba1-final-de-verdad".

Tu primer programa, pieza por pieza

Vamos a desarmar el programa "Hello, World!" como un mecánico desarma un motor: pieza por pieza, hasta entender qué hace cada parte. Usaremos la versión clásica con método Main, porque te enseña la estructura completa; después veremos la versión simplificada (top-level statements) y entenderás que es solo un atajo.

Aquí está el programa completo otra vez, numerado para referencia:

```
1  using System;
2
3  namespace MiPrimerPrograma;
4
5  class Program
6  {
7      static void Main(string[] args)
8      {
9          Console.WriteLine("Hello, World!");
10     }
11 }
```

Línea 1: "using System;". La palabra "using" le dice al compilador: "voy a usar cosas que viven en el namespace System, así que no me obligues a escribir el nombre completo cada vez". Es como decirle al cartero "vivo en la colonia Centro" para no tener que dictarle la dirección completa en cada carta. El namespace System es la biblioteca fundamental de .NET: ahí viven Console, String, Math y cientos de tipos básicos. Sin esta línea, tendrías que escribir "System.Console.WriteLine" en lugar de "Console.WriteLine". Funciona igual, pero es más tedioso. Casi todos los programas C# empiezan con uno o varios "using".

Línea 3: "namespace MiPrimerPrograma;". Un namespace (espacio de nombres) es un apellido organizativo para tu código. Imagina que en tu escuela hay dos alumnos llamados "Juan": para no confundirlos, usas "Juan Pérez" y "Juan García". Igual en programación: puede haber dos clases llamadas "Program" en bibliotecas distintas, y el namespace evita choques. Tu proyecto crea su propio namespace con el nombre del proyecto. El punto y coma al final es la sintaxis moderna ("file-scoped namespace", introducida en C# 10): significa "todo este archivo pertenece a este namespace", sin necesidad de encerrar el código entre llaves extra. Es un ejemplo de cómo C# moderno elimina ceremonia innecesaria.

Línea 5: `"class Program"`. Todo en C# vive dentro de clases (ya profundizaremos en la Unidad de programación orientada a objetos, pero adelanto la idea). Una clase es un molde o plantilla que agrupa datos y acciones. Por ahora piensa en `"class Program"` como la caja que contiene tu programa: el punto de entrada tiene que vivir en algún lado, y ese lado es una clase. El nombre `"Program"` es convención: la plantilla lo usa siempre.

Línea 7: `"static void Main(string[] args)"`. Esta es la línea más importante del programa: el método `Main` es el punto de entrada, el lugar por donde la ejecución EMPIEZA. Cuando haces `"dotnet run"`, el CLR busca el método `Main` y empieza a ejecutar sus instrucciones de arriba hacia abajo. Desmenucemos cada palabra:

- `"static"` significa que este método pertenece a la clase misma y no necesita que se cree un objeto para llamarlo. El CLR necesita arrancar el programa sin tener nada creado todavía, así que el punto de entrada tiene que ser estático. (No te preocupes si `"objeto"` aún no significa mucho: lo verás a fondo más adelante. Por ahora: `static` = se puede llamar sin crear nada antes.)
- `"void"` significa que este método no devuelve ningún valor. Es como una receta que sigues por sus efectos (imprimir en pantalla) y no porque te entregue un plato al final.
- `"Main"` es el nombre mágico que el CLR busca. Tiene que escribirse exactamente así, con M mayúscula: C# distingue entre mayúsculas y minúsculas, y `"main"` no es lo mismo que `"Main"`.
- `"(string[] args)"` son los parámetros: `"args"` es un arreglo de textos con los argumentos que alguien le pase a tu programa desde la terminal. Por ejemplo, si ejecutas tu programa con `"dotnet run -- Hola"`, la palabra `"Hola"` llega dentro de `args`. Al empezar no lo usarás, pero es bueno saber que existe.

Líneas 8 y 10: las llaves `"{"` y `"}"`. En C#, las llaves delimitan bloques de código: todo lo que está entre la llave de apertura y la de cierre pertenece a ese método (o clase, o condición). La indentación (los espacios al inicio) no es obligatoria para el compilador, pero es sagrada para los humanos: el código bien indentado se lee de un vistazo; el código sin indentar es un laberinto. VS Code la maneja automáticamente: no pelees con ella, déjala trabajar.

Línea 9: `Console.WriteLine("Hello, World!");`. La estrella del show. "Console" representa la terminal (la ventana de texto donde corre tu programa). "WriteLine" es un método que escribe algo y luego salta de línea (Write a secas escribe sin saltar). Entre paréntesis va lo que quieres mostrar: "Hello, World!" entre comillas dobles es un string, una cadena de texto. Y el punto y coma final es obligatorio en C#: marca el fin de la instrucción, como el punto al final de una oración. Olvidar un punto y coma es el error número uno de los principiantes en C#, y el compilador te lo señalará con un mensaje de error. Acostúmbrate: en C#, cada instrucción termina con ";".

Top-level statements: la versión simplificada

Las versiones modernas de C# (9 en adelante) permiten escribir programas de consola sin toda la ceremonia de namespace, clase y Main. A esto se le llama "top-level statements" (instrucciones de nivel superior): simplemente escribes las instrucciones directamente en el archivo, y el compilador genera por ti, entre bambalinas, la clase Program y el método Main.

```
Console.WriteLine("Hello, World!");
```

Eso es todo el programa. Una línea. El compilador entiende que esas instrucciones son el cuerpo del Main. Además, las plantillas modernas incluyen "usings implícitos": el "using System;" ya viene activado por configuración del proyecto (en el .csproj), así que ni siquiera necesitas escribirlo para lo básico.

En este libro usaremos top-level statements en los ejemplos cortos, porque nos dejan enfocarnos en lo que estamos aprendiendo sin distracciones. Pero es importantísimo que entiendas la versión completa con Main, por tres razones: (1) muchísimo código existente y tutoriales usan la versión clásica; (2) cuando tus programas crezcan, necesitarás clases y métodos de verdad; (3) en entrevistas de trabajo y exámenes, esperan que sepas qué es el método Main. Piensa en los top-level statements como el modo "automático" del coche: comodísimo para el día a día, pero debes saber manejar estándar por si te toca.

Comentarios: notas para humanos

Los comentarios son texto que escribes en el código para explicarlo, y que el compilador ignora por completo. Son notas para ti del futuro y para otros programadores. En C# hay dos formas:

```
// Esto es un comentario de una línea: empieza con doble diagonal

/*
    Esto es un comentario de varias líneas:
    empieza con diagonal-asterisco y termina
    con asterisco-diagonal. Útil para explicaciones largas
    o para desactivar temporalmente un bloque de código.
*/
```

Buenas prácticas con comentarios (esto separa a los novatos de los profesionales): comenta el POR QUÉ, no el QUÉ. Un comentario que dice "// suma a y b" encima de una suma no aporta nada (el código ya dice eso); un comentario que dice "// se suma el IVA porque la factura lo requiere por ley" explica algo que el código no dice. Además: no dejes código comentado muerto pudriéndose en tus archivos (bórralo, para eso existen los sistemas de control de versiones como Git), mantén los comentarios actualizados (un comentario mentiroso es peor que ningún comentario), y escribe los comentarios en el mismo idioma que tu equipo (en este libro, español).

Errores comunes de principiantes (y cómo evitarlos)

1. Olvidar el punto y coma. El compilador dirá algo como "; expected". Solución: respira, ve a la línea que indica el error (a veces es la línea anterior) y agrega el ";". Con una semana de práctica, tus dedos lo pondrán en automático.
2. Confundir mayúsculas y minúsculas. "console.writeline" no existe; es "Console.WriteLine" con C y W mayúsculas. C# es case-sensitive (sensible a mayúsculas). El autocompletado de VS Code es tu mejor amigo: si lo usas, casi nunca tendrás este problema.
3. Comillas incorrectas. Los textos van con comillas dobles ("hola"). Las comillas simples ('h') son para un solo carácter (tipo char, lo verás en la Unidad 2). Mezclarlas produce errores.
4. No guardar antes de ejecutar. Si editas Program.cs y corres "dotnet run" sin guardar, se ejecuta la versión vieja y jurarás que tu cambio "no funciona". Guarda siempre (Ctrl+S) antes de ejecutar. VS Code puede configurarse para autoguardar: actívalo.
5. Ejecutar desde la carpeta equivocada. "dotnet run" debe ejecutarse dentro de la carpeta del proyecto (donde está el .csproj). Si lo corres desde otra carpeta, verás un error de que no encuentra el proyecto.
6. Miedo a los mensajes de error. Los mensajes del compilador de C# son, en general, buenos y te dicen la línea y el problema. Léelos completos en lugar de entrar en pánico: nueve de cada diez veces la solución está escrita ahí mismo.

Buenas prácticas profesionales desde el día uno

- Nombres descriptivos: aunque hoy tu programa es de una línea, acostumbra a poner nombres que expliquen. Ya lo practicarás con variables en la Unidad 2.
- Un proyecto por ejercicio: no mezcles veinte experimentos en un solo Program.cs; crea proyectos pequeños y enfocados.
- Lee los errores de abajo hacia arriba: a veces un solo error temprano genera una cascada de errores falsos. Corrige el primero y recompila.
- Usa la terminal sin miedo: "dotnet new", "dotnet build", "dotnet run" son tus herramientas diarias. Los profesionales viven en la terminal; mientras antes te sientas cómodo, mejor.
- Versiona tu código con Git cuando empieces a hacer proyectos que te importen (lo veremos más adelante): es la red de seguridad de todo programador.

El ciclo de desarrollo profesional: escribir, compilar, ejecutar, depurar

Todo programador profesional, sin importar el lenguaje, repite un ciclo decenas de veces al día: escribe un poco de código, lo compila, lo ejecuta, observa qué pasa, y depura (encuentra y corrige errores). En C# con .NET, ese ciclo se ve así en la práctica, y vale la pena que lo interiorices desde hoy porque es tu rutina de trabajo de ahora en adelante.

Paso 1: escribir. Abres VS Code en la carpeta de tu proyecto y modificas Program.cs. Mientras escribes, la extensión de C# ya está trabajando: subraya en rojo lo que está mal, te sugiere métodos con IntelliSense y hasta te propone arreglos automáticos (el foquito que aparece al lado de los errores). Aprovecha esto: es como escribir con un corrector ortográfico que además sabe gramática. No esperes a ejecutar para descubrir un error de dedo; si ves rojo, atiéndelo.

Paso 2: compilar. Guardas (Ctrl+S) y en la terminal ejecutas "dotnet build". El compilador traduce tu C# a IL y te reporta errores y advertencias. Aprende a distinguirlos: los errores (errors) impiden que el programa se genere; las advertencias (warnings) permiten compilar pero te avisan de algo sospechoso (una variable que declaraste y nunca usaste, por ejemplo). Buena práctica profesional: trata las advertencias como errores. Un proyecto limpio compila con cero errores y cero advertencias. Los equipos serios configuran sus proyectos para que las advertencias se traten como errores automáticamente.

Paso 3: ejecutar. Con "dotnet run" compilas (si hubo cambios) y ejecutas. Observa la salida con ojo crítico: ¿es EXACTAMENTE lo que esperabas? Los principiantes suelen ver que "algo salió" y darlo por bueno; los profesionales comparan carácter por carácter contra lo esperado. Si algo no cuadra, no cambies código al azar: primero entiende.

Paso 4: depurar. Depurar (debugging) es el arte de encontrar por qué tu programa no hace lo que esperas. Las técnicas, de la más simple a la más avanzada, son: (a) leer el código en voz alta como si fueras la computadora, línea por línea; (b) agregar `Console.WriteLine` temporales que muestren valores intermedios ("¿qué vale esta variable aquí?"); (c) leer el mensaje de error completo, incluyendo el número de línea y el tipo de excepción; (d) usar el depurador de VS Code (F5), que te permite ejecutar línea por línea, pausar en puntos de interrupción (breakpoints) e inspeccionar el valor de cada variable en cada momento. El depurador suena intimidante al principio, pero es la herramienta más poderosa que tienes: cuando la domines, encontrar errores dejará de ser adivinanza para convertirse en investigación.

Un hábito que te va a ahorrar horas: cuando algo falla, formula una hipótesis antes de tocar el código ("creo que el problema es que esta variable vale cero aquí"), y luego compruébala (imprime la variable o pon un breakpoint). Cambiar código al azar con la esperanza de que "algo funcione" es la forma más lenta de programar que existe. Los programadores senior no son los que menos errores cometen: son los que más rápido los encuentran, porque tienen método.

C# frente a otros lenguajes: el mapa completo

En la sección anterior comparamos C# de pasada con otros lenguajes. Vamos a profundizar, porque elegir bien tu primer lenguaje (o tu segundo) es una decisión de carrera, y quiero que la tomes con información, no con modas.

C# vs Python. Python es el lenguaje favorito para empezar por su sintaxis minimalista y es el rey de la ciencia de datos, la inteligencia artificial y los scripts rápidos. Pero Python es interpretado y de tipado dinámico: los errores de tipo aparecen en ejecución, y su rendimiento es mucho menor (aunque para la mayoría de scripts no importa). C# te exige más ceremonia al inicio (tipos, punto y coma, compilación), pero a cambio te da velocidad de ejecución muy superior, detección temprana de errores y un ecosistema más robusto para aplicaciones grandes. Analogía: Python es una bicicleta (ágil, fácil, perfecta para distancias cortas); C# es un automóvil (requiere licencia y mantenimiento, pero te lleva lejos con carga pesada). Muchos programadores aprenden ambos: Python para experimentar rápido, C# para construir sistemas serios.

C# vs JavaScript. JavaScript es obligatorio si quieres programar para el navegador: no hay alternativa. Con Node.js también se usa en el servidor. Su ecosistema es gigantesco pero caótico (frameworks que nacen y mueren cada año), y su tipado dinámico genera una categoría famosa de bugs sutiles. C# con ASP.NET Core compite en el backend con mejor rendimiento, tipado fuerte y herramientas más estables. En la práctica, muchísimos desarrolladores full-stack combinan ambos: JavaScript/TypeScript en el frontend, C# en el

backend. TypeScript, por cierto, fue creado por el mismo Anders Hejlsberg: el ADN de C# también corre por ahí.

C# vs Java. Este es el duelo clásico: son primos hermanos (misma filosofía, sintaxis parecida, ambos con máquina virtual y recolección de basura). Java domina en grandes corporaciones antiguas, en Android (aunque Kotlin le come terreno) y en sistemas legacy. C# suele tener un lenguaje más moderno (evoluciona más rápido), mejores herramientas gratuitas (el ecosistema de Java depende más de herramientas de terceros) y la ventaja de Unity y el mundo Microsoft. En el mercado laboral hispanohablante y en Estados Unidos, ambos tienen demanda sólida; la elección muchas veces depende de qué empresas hay en tu ciudad. Mi consejo de colega: domina bien uno de los dos y el otro lo aprendes en semanas cuando lo necesites, porque el 80 por ciento de los conceptos se transfieren.

C# vs C++. C++ te da control total del hardware y el máximo rendimiento posible: motores de videojuegos AAA, sistemas operativos, software embebido. El precio: gestión manual de memoria (una fuente inagotable de errores graves), compilaciones lentas y una curva de aprendizaje brutal. C# sacrifica un poco de control y rendimiento a cambio de muchísima productividad y seguridad (el CLR y la recolección de basura te cuidan). Para el 95 por ciento de las aplicaciones, C# es más que suficientemente rápido. Aprende C++ solo si tu camino apunta a gráficos de bajo nivel, motores o sistemas embebidos; para casi todo lo demás, C# te rinde más.

C# vs Go y Rust (los modernos). Go brilla en microservicios y herramientas de infraestructura por su simplicidad; Rust brilla donde se necesita seguridad de memoria SIN recolección de basura (sistemas, navegadores). Ambos son excelentes, pero tienen ecosistemas más jóvenes y mercados laborales más pequeños que C#. C# es la apuesta equilibrada: moderno pero maduro, con empleo abundante.

La conclusión honesta: no existe "el mejor lenguaje", existe el mejor lenguaje para cada trabajo y cada etapa. C# está en el punto dulce para ti ahora: te enseña disciplina de tipos (que te servirá en cualquier lenguaje), tiene curva de aprendizaje razonable, herramientas gratuitas de nivel profesional, y te abre puertas en videojuegos, empresas, web y nube. Cuando domines C#, aprender tu segundo lenguaje será mucho más fácil, porque ya pensarás como programador.

Publicar tu programa: del código al ejecutable

Hasta ahora ejecutas tu programa con "dotnet run", que es la forma de DESARROLLAR. Pero ¿qué pasa cuando quieres compartir tu programa con alguien que no tiene el SDK instalado? Para eso existe "dotnet publish": empaqueta tu programa en una carpeta lista para distribuir.

Pruébalo con tu proyecto de ejemplo:

```
$ dotnet publish -c Release
```

El parámetro "-c Release" indica que quieres la versión optimizada (Release), no la de desarrollo (Debug, que incluye información extra para depurar y es más lenta). Al terminar, encontrarás tu programa en una ruta como "bin/Release/net8.0/publish/". Ahí verás tu ejecutable y sus dependencias. En Windows será un .exe que puedes copiar a otra computadora y ejecutar con doble clic (necesita el runtime de .NET instalado, o puedes publicar en modo "self-contained" para incluirlo todo, aunque el archivo resultante es más grande).

Existen dos modos de publicación que vale la pena conocer desde ahora:

- Dependiente del framework (framework-dependent, el modo por defecto): tu programa es ligero, pero la computadora destino necesita tener instalado el runtime de .NET de la versión correspondiente. Ideal para servidores y entornos controlados.
- Autocontenido (self-contained): empaqueta el runtime junto con tu programa. El resultado es más pesado (decenas de megabytes), pero corre en cualquier máquina compatible sin instalar nada. Ideal para distribuir a usuarios finales.

Para publicar autocontenido para Windows desde cualquier sistema, por ejemplo:

```
$ dotnet publish -c Release -r win-x64 --self-contained
```

El "-r win-x64" especifica el "runtime identifier": la plataforma destino (win-x64 para Windows de 64 bits, linux-x64 para Linux, osx-x64 para macOS). Esto es la magia multiplataforma en acción: compilas en tu sistema y generas el ejecutable para otro.

No necesitas dominar la publicación hoy; lo importante es que sepas que existe y que el camino completo es: desarrollar con "dotnet run", verificar con "dotnet build", y distribuir con "dotnet publish". Cuando en unidades futuras construyas programas que sí quieras compartir (una calculadora, un juego de consola), este comando será tu puerta de salida al mundo real.

Preguntas frecuentes de quien empieza (respóndelas antes de que te frenen)

Cuando alguien empieza con C#, las mismas dudas aparecen una y otra vez. Las respondo aquí de frente para que ninguna te detenga a las 2 de la mañana.

"¿Necesito una computadora potente?" No. Para todo este libro basta cualquier computadora de los últimos 10 años con 4 GB de RAM. El .NET SDK y VS Code son ligeros comparados

con el Visual Studio completo. Si tu máquina puede navegar por internet sin sufrir, puede compilar C#.

"¿Windows, Mac o Linux?" El que tengas. .NET moderno funciona igual de bien en los tres, y los comandos "dotnet" son idénticos. Los ejemplos de este libro están probados mentalmente en los tres sistemas. La única diferencia menor: en Windows las rutas usan diagonal invertida y en Mac/Linux diagonal normal, pero eso no afecta tu código C#.

"¿Cuánto tiempo me tomará aprender C#?" Depende de tu constancia, no de tu talento. Con una hora diaria de práctica real (escribiendo código, no solo leyendo), en 2-3 meses tendrás bases sólidas para construir programas de consola completos y entender código ajeno. En 6-12 meses puedes estar listo para un empleo junior si además aprendes un área de aplicación (web con ASP.NET, juegos con Unity, etc.). Desconfía de quien prometa "aprende a programar en 7 días": la programación es una habilidad, como tocar un instrumento, y las habilidades se construyen con repetición.

"¿Tengo que saber matemáticas?" Para el 90 por ciento de la programación, no. Necesitas lógica (si esto, entonces aquello), no cálculo. Los videojuegos con físicas 3D y la ciencia de datos sí piden matemáticas, pero el desarrollo de aplicaciones empresariales, web y la mayoría del software usan aritmética básica. Si reprobaste matemáticas en la escuela, no es impedimento: la lógica de programación es una habilidad distinta.

"¿Y si me equivoco de carrera eligiendo C#?" No existe la elección irreversible en programación. Los conceptos que aprendas (variables, condiciones, ciclos, objetos, tipos) son universales; cambiar de lenguaje después es cuestión de semanas, no de años. Además, C# está entre los lenguajes con más demanda laboral del mundo desde hace dos décadas: es una de las apuestas más seguras que puedes hacer.

"¿Debo memorizar todo?" No, y nadie lo hace. Los profesionales consultan documentación todos los días. Lo que sí debes interiorizar son los conceptos y el flujo de trabajo; la sintaxis exacta la refuerza la práctica y el autocompletado. Si dentro de un mes no recuerdas si era "WriteLine" o "Writeline", el IntelliSense te lo recordará en medio segundo. Memoriza ideas, no letras.

Ejemplos paso a paso

La teoría sin práctica es como aprender a nadar leyendo un manual: entiendes la idea, pero en el agua te hundes. En esta sección vamos a escribir varios programas completos, del más simple al más interesante, y después de cada bloque de código lo vamos a explicar línea por línea. Te pido un favor profesional: no te limites a leerlos. Crea un proyecto nuevo para cada

ejemplo ("dotnet new console -n Ejemplo1", etc.), escribe el código a mano (no copiar-pegar) y ejecútalo con "dotnet run". Escribir a mano es lo que entrena tus dedos y tus ojos a detectar errores.

Ejemplo 1: El clásico "Hola, mundo" personalizado

Empecemos con lo más básico: un programa que saluda. Parece trivial, pero aquí practicas el flujo completo (crear, escribir, guardar, ejecutar) y confirmas que tu instalación funciona.

```
Console.WriteLine("Hola, mundo!");  
Console.WriteLine("Estoy aprendiendo C# en Lucas EGA Academy.");
```

Explicación línea por línea:

- Línea 1: "Console.WriteLine("Hola, mundo!");" - Llama al método WriteLine de la clase Console para mostrar el texto "Hola, mundo!" en la terminal. Fíjate en las tres partes: "Console" (dónde), "WriteLine" (qué acción, con W y L mayúsculas), y entre paréntesis y comillas dobles el texto a mostrar. El punto y coma cierra la instrucción.
- Línea 2: otra instrucción WriteLine que imprime la segunda línea. WriteLine, a diferencia de Write, después de imprimir salta a la siguiente línea, por eso cada mensaje aparece en su propio renglón.

Al ejecutar verás:

```
Hola, mundo!  
Estoy aprendiendo C# en Lucas EGA Academy.
```

Experimento para ti: cambia WriteLine por Write en ambas líneas y ejecuta de nuevo. Verás que los dos mensajes salen pegados en el mismo renglón. Así entiendes la diferencia entre Write (escribe sin saltar) y WriteLine (escribe y salta). Este tipo de micro-experimentos es como aprenden los programadores de verdad: cambiando una cosa y observando.

Ejemplo 2: Múltiples líneas y caracteres especiales

Ahora juguemos con el formato: saltos de línea dentro del texto, comillas dentro de comillas y caracteres especiales. Para eso C# usa "secuencias de escape": combinaciones que empiezan con diagonal invertida () y representan caracteres difíciles de escribir directamente.

```
Console.WriteLine("Primera línea\nSegunda línea");  
Console.WriteLine("Ella dijo: \"Me encanta C#\"");  
Console.WriteLine("Ruta de ejemplo: C:\\proyectos\\csharp");  
Console.WriteLine("Columnas:\tNombre\tEdad");
```

Explicación línea por línea:

- Línea 1: "\n" dentro del texto significa "salto de línea". Aunque escribiste una sola instrucción, en pantalla aparecen dos líneas. Es muy útil cuando quieres controlar el formato sin escribir cien WriteLine.
- Línea 2: \" permite poner comillas dobles DENTRO de un texto delimitado por comillas dobles. Sin la diagonal, el compilador creería que el texto termina antes de tiempo y marcaría error. Esto se llama "escapar" un carácter.
- Línea 3: "\\" representa una sola diagonal invertida. Las rutas de Windows usan diagonales invertidas, así que para mostrarlas hay que escaparlas. (Alternativa moderna: usar "verbatim strings" con arroba, como @"C:\proyectos\csharp", donde no necesitas escapar nada. Lo veremos en la unidad de cadenas.)
- Línea 4: "\t" es un tabulador: alinea el texto en columnas. Útil para mostrar datos ordenados en la consola.

Salida:

```
Primera línea  
Segunda línea  
Ella dijo: "Me encanta C#"  
Ruta de ejemplo: C:\proyectos\csharp  
Columnas:      Nombre  Edad
```

Ejemplo 3: Tu primer programa "útil" - una tarjeta de presentación

Vamos a hacer algo que se sienta real: un programa que imprime una tarjeta de presentación con tus datos, usando líneas decorativas. Aquí practicas planear la salida antes de escribir el código, que es una habilidad profesional clave.

```

Console.WriteLine("=====");
Console.WriteLine("  LUCAS EGA ACADEMY");
Console.WriteLine("  Curso Completo de C#");
Console.WriteLine("=====");
Console.WriteLine("Estudiante: [Tu nombre]");
Console.WriteLine("Meta: dominar C# y .NET");
Console.WriteLine("Primera unidad: ¡completada!");

```

Explicación:

- Las líneas 1 y 4 imprimen una fila de signos "=" para crear un encabezado visual. Es un truco viejísimo de la programación de consola y sigue usándose para que la salida se vea ordenada.
- Las líneas 2 y 3 llevan dos espacios al inicio para centrar visualmente el texto dentro del marco. No es centrado matemático, es centrado "a ojo", y está bien: en consola la estética es aproximada.
- Las líneas 5 a 7 muestran datos. Cambia "[Tu nombre]" por tu nombre real y ejecuta: ver un programa que habla de TI es motivador, y personalizar ejemplos es un hábito que te recomiendo para todo el libro.

Salida:

```

=====
  LUCAS EGA ACADEMY
  Curso Completo de C#
=====
Estudiante: [Tu nombre]
Meta: dominar C# y .NET
Primera unidad: ¡completada!

```

Fíjate que "¡" y "¿" funcionan sin problema en la consola moderna. Si algún día ves símbolos raros en lugar de acentos, es un tema de codificación de la terminal (en Windows antiguo a veces hay que ejecutar "chcp 65001" para UTF-8), no de tu código.

Ejemplo 4: Comentarios en acción y estructura clásica con Main

Para cerrar los ejemplos, vamos a escribir el mismo saludo pero con la estructura clásica completa (namespace, clase, Main) y comentarios que expliquen cada parte. Este ejemplo te sirve de plantilla mental: cuando veas código con esta estructura en internet, ya no te asustará.

```
// Programa: SaludoClásico
// Autor: yo, estudiante de C#
// Descripción: muestra un saludo usando la estructura clásica con Main

using System;

namespace SaludoClasico;

class Program
{
    // El método Main es el punto de entrada del programa.
    // La ejecución empieza aquí, en la primera línea del método.
    static void Main(string[] args)
    {
        Console.WriteLine("Hola desde el método Main.");
        Console.WriteLine("La ejecución empezó aquí arriba y termina aquí abajo.");
    }
}
```

Explicación línea por línea:

- Líneas 1-3: comentarios de una línea con información del programa. Buena costumbre en programas que vas a guardar o compartir.
- Línea 5: "using System;" importa el namespace fundamental para poder usar Console sin escribir "System.Console".
- Línea 7: declaración del namespace del proyecto (file-scoped, con punto y coma al final).
- Línea 9: declaración de la clase contenedora "Program".
- Líneas 10 y 14: llaves que delimitan el cuerpo de la clase.
- Línea 13: "static void Main(string[] args)" declara el punto de entrada. Recuerda: static (sin crear objetos), void (no devuelve nada), Main (nombre exacto con mayúscula), args (argumentos de terminal, hoy sin usar).
- Líneas 11-12: comentarios dentro del método explicando el flujo.
- Líneas 15-16: las dos instrucciones que se ejecutan, en orden, de arriba hacia abajo. El programa termina cuando se ejecuta la última línea del Main.

Para crear este ejemplo, el comando sería "dotnet new console -n SaludoClasico" y luego reemplazarías el contenido de Program.cs con este código. Al hacer "dotnet run" verás las dos líneas del saludo. Compáralo con el Ejemplo 1 (top-level statements): mismo resultado, más ceremonia. Ahora entiendes las dos caras de C# moderno.

Ejemplo 5: Leyendo el flujo - qué pasa si cambio el orden

Este último ejemplo no introduce sintaxis nueva: introduce una forma de PENSAR. La computadora ejecuta tus instrucciones en orden, de arriba hacia abajo, y el orden importa muchísimo.

```
Console.WriteLine("Paso 1: me levanto");  
Console.WriteLine("Paso 2: me lavo los dientes");  
Console.WriteLine("Paso 3: desayuno");  
Console.WriteLine("Paso 4: salgo a trabajar");
```

Si intercambias las líneas, el "programa" cuenta una historia absurda (salir a trabajar antes de levantarse). Parece obvio, pero este principio - el flujo secuencial - es la base de todo: más adelante, las condiciones (if) y los ciclos (for, while) son formas de alterar ese orden, de decidir qué líneas se ejecutan y cuántas veces. Por ahora quédate con la idea: el programa hace EXACTAMENTE lo que escribes, en el orden en que lo escribes, ni más ni menos. Cuando un programa "no hace lo que quiero", el 99 por ciento de las veces es porque lo que escribiste no es lo que creías haber escrito. Leer tu código en voz alta, línea por línea, como si fueras la computadora, es la técnica de depuración más antigua y efectiva que existe.

Práctica

Aquí es donde el aprendizaje se vuelve real. Te presento 8 ejercicios ordenados de menor a mayor dificultad. Primero intenta resolverlos TÚ, sin mirar las soluciones: crea un proyecto nuevo por ejercicio, escribe el código, ejecútalo y verifica que la salida sea la esperada. Solo cuando lo hayas intentado de verdad (o te hayas atorado más de 15 minutos) consulta la solución. Copiar la solución sin intentarlo es como ver a alguien hacer ejercicio por ti: no desarrollas músculo.

1. Mi primera salida. Crea un proyecto llamado "Ejercicio1" que imprima en la consola tu nombre completo en una línea y tu ciudad en otra línea. Usa dos instrucciones WriteLine.
2. Write vs WriteLine. Crea un proyecto llamado "Ejercicio2" que muestre las palabras "Rojo", "Verde" y "Azul". Primero imprímelas cada una en su propia línea usando WriteLine, y después imprímelas las tres en la misma línea usando Write. Observa la diferencia en la salida.
3. Arte ASCII. Crea un proyecto llamado "Ejercicio3" que dibuje en la consola una carita feliz usando caracteres de texto, algo así:

```

*****
*       *
*  O O  *
*   ^   *
*  \___/ *
*       *
*****

```

No tiene que ser idéntica: lo importante es que uses varios `WriteLine` para componer el dibujo línea por línea. (Pista: para imprimir la diagonal invertida necesitas escribir "\\").)

4. Mi tarjeta de presentación completa. Crea un proyecto llamado "Ejercicio4" que imprima una tarjeta con marco de asteriscos que incluya: tu nombre, tu ocupación o lo que estudias, tu lenguaje favorito (debería ser C# a estas alturas) y una meta para este año. Usa secuencias de escape donde las necesites.
5. Estructura clásica. Crea un proyecto llamado "Ejercicio5". Borra el contenido generado de `Program.cs` y escribe desde cero (sin copiar-pegar) la estructura clásica completa: `using System`, `namespace Ejercicio5` (file-scoped), clase `Program` y método `Main` que imprima tres líneas sobre lo que más te gustó de esta unidad. Luego compíllalo y ejecútalo. El objetivo es que tus dedos aprendan la estructura de memoria.
6. El narrador. Crea un proyecto llamado "Ejercicio6" que cuente una micro-historia de 6 líneas usando `WriteLine`, donde uses al menos tres secuencias de escape distintas (`\n`, `\t`, `\"` o `\\`) con un propósito claro en la historia (un diálogo con comillas, una tabulación, un salto de línea intencional). Agrega al inicio un comentario de una línea con el título de tu historia.
7. Depurador. El siguiente programa tiene 4 errores (de compilación o de lógica). Cópialo a un proyecto llamado "Ejercicio7", encuentra y corrige cada error, y escribe un comentario encima de cada línea corregida explicando qué estaba mal:

```

console.WriteLine("Empezando el programa...")
Console.WriteLine('Segunda línea');
Console.WriteLine("Tercera línea")
Console.Writeline("Cuarta línea");

```

8. Mi menú de consola. Crea un proyecto llamado "Ejercicio8" que simule el menú de un programa real, con este formato exacto (adapta el título):

```
=====
MI PROGRAMA v1.0
=====
1. Saludar
2. Despedirse
3. Salir
=====
Elige una opción:
```

Usa Write para la última línea ("Elige una opción: ") de modo que el cursor quede esperando en la misma línea, como haría un programa real. Agrega un comentario multilínea al inicio explicando qué hace el programa.

Soluciones

Solución 1: Mi primera salida.

```
Console.WriteLine("Esteban Rivera");
Console.WriteLine("Ciudad de México");
```

Explicación: dos instrucciones WriteLine, cada una imprime un texto y salta de línea. Cambia los datos por los tuyos. Este ejercicio valida tu flujo completo: crear el proyecto, editar Program.cs, guardar y ejecutar con "dotnet run". Si viste tu nombre en la terminal, tu entorno funciona al cien.

Solución 2: Write vs WriteLine.

```
Console.WriteLine("Rojo");
Console.WriteLine("Verde");
Console.WriteLine("Azul");
Console.Write("Rojo ");
Console.Write("Verde ");
Console.Write("Azul");
```

Explicación: las primeras tres líneas usan WriteLine y cada color sale en su renglón. Las siguientes tres usan Write, que NO salta de línea, así que los tres colores salen pegados en un solo renglón (agregué un espacio al final de "Rojo " y "Verde " para que no salgan amontonados). La lección: WriteLine = escribir + salto; Write = escribir sin salto. Parece un detalle menor, pero en programas reales controlar exactamente dónde salta la línea es fundamental para menús y reportes.

Solución 3: Arte ASCII.

```
Console.WriteLine(" *****");
Console.WriteLine(" *           *");
Console.WriteLine("*   0 0   *");
Console.WriteLine("*     ^     *");
Console.WriteLine("*  \\___/  *");
Console.WriteLine(" *           *");
Console.WriteLine(" *****");
```

Explicación: cada WriteLine dibuja una fila del dibujo. El punto delicado está en la línea de la boca: para mostrar "___/" hay que escribir "___/", porque "\\" es la secuencia de escape que representa una sola diagonal invertida. Si hubieras escrito "___/" directamente, el compilador marcaría error porque "_" no es una secuencia de escape válida. El arte ASCII es un excelente ejercicio de precisión: un espacio de más o de menos y el dibujo se deforma, igual que un punto y coma de más o de menos rompe un programa.

Solución 4: Mi tarjeta de presentación completa.

```
Console.WriteLine("*****");
Console.WriteLine("*   Esteban Rivera       *");
Console.WriteLine("*   Estudiante de C#     *");
Console.WriteLine("*   Lenguaje favorito: C# *");
Console.WriteLine("*   Meta 2026: primer    *");
Console.WriteLine("*   empleo como dev     *");
Console.WriteLine("*****");
```

Explicación: el marco se construye con asteriscos y los datos van con espacios para alinear el borde derecho. No es necesario que quede perfecto al milímetro; lo valioso es que practicaste planear una salida con formato antes de escribir código, que es lo que harás con reportes y menús toda tu carrera. Si quisiste poner comillas en algún dato, recuerda escaparlas con \".

Solución 5: Estructura clásica.


```

using System;

namespace Ejercicio5;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Lo que más me gustó fue entender el método Main.");
        Console.WriteLine("Ahora sé por dónde empieza a ejecutarse mi programa.");
        Console.WriteLine("Y ya sé crear proyectos con dotnet new console.");
    }
}

```

Explicación: aquí está la estructura completa escrita a mano. Repasa cada parte: "using System;" (importa lo básico), "namespace Ejercicio5;" (tu espacio de nombres, con punto y coma por ser file-scoped), "class Program" (la caja contenedora), "static void Main(string[] args)" (el punto de entrada: static porque arranca sin objetos, void porque no devuelve nada, Main con mayúscula porque C# distingue mayúsculas). Si lo escribiste sin mirar y compiló a la primera, tus dedos ya saben más C# del que crees.

Solución 6: El narrador.

```

// Historia: El programador y la taza de café
Console.WriteLine("Eran las 11 de la noche.\nEl programador miró su taza vacía.");
Console.WriteLine("Suspiró y dijo: \"Necesito más café para terminar este bug.\"");
Console.WriteLine("Pasos para sobrevivir:\t1. Café\t2. Código\t3. Dormir");
Console.WriteLine("Fin de la historia.\\"Continuará...");

```

Explicación: el comentario de una línea pone título a la historia. Luego: "\n" parte la primera línea en dos para dar ritmo narrativo; "\"" permite el diálogo con comillas dentro del texto; "\t" alinea los tres pasos como columnas; "\\" muestra una diagonal invertida literal antes de "Continuará". Cada secuencia de escape tiene un propósito narrativo, no está de adorno: esa es la diferencia entre usar una herramienta y entenderla.

Solución 7: Depurador. El código corregido queda así:

```

Console.WriteLine("Empezando el programa..."); // Error 1: "console" debía ser
"Console" (mayúscula; C# distingue mayúsculas)
Console.WriteLine("Segunda línea"); // Error 2: las comillas simples son para un
char; los textos usan comillas dobles
Console.WriteLine("Tercera línea"); // Error 3: faltaba el punto y coma al final de
la instrucción
Console.WriteLine("Cuarta línea"); // Error 4: "Writeline" debía ser "WriteLine" (L
mayúscula)

```

Explicación: los cuatro errores son los cuatro clásicos del principiante en C#. Error 1 y 4: C# es case-sensitive, las mayúsculas importan y el autocompletado de VS Code te salva de esto. Error 2: comillas simples vs dobles (lo verás a fondo en la Unidad 2 con char vs string). Error 3: el punto y coma olvidado, el error más común de todos. Si encontraste los cuatro sin ayuda, ya piensas como depurador.

Solución 8: Mi menú de consola.

```

/*
Programa: Mi menú de consola
Descripción: muestra un menú con formato de programa real.
La última línea usa Write (sin salto) para que el cursor
quede esperando la opción en el mismo renglón.
*/

Console.WriteLine("=====");
Console.WriteLine("  MI PROGRAMA v1.0");
Console.WriteLine("=====");
Console.WriteLine("  1. Saludar");
Console.WriteLine("  2. Despedirse");
Console.WriteLine("  3. Salir");
Console.WriteLine("=====");
Console.Write("Elige una opción: ");

```

Explicación: el comentario multilínea documenta el programa (observa que explica el POR QUÉ de la decisión de usar Write al final, no solo el qué). El menú usa líneas decorativas y dos espacios de indentación para las opciones, imitando programas reales de consola. La última línea usa Write a propósito: en un programa real, después de "Elige una opción: " el usuario escribiría su elección en ese mismo renglón (en la Unidad 2 aprenderás a LEER esa elección con Console.ReadLine, y este menú cobrará vida de verdad).

Resumen de la unidad

- C# (pronunciado "si sharp") es un lenguaje moderno, orientado a objetos y de tipado fuerte, creado por Microsoft y diseñado originalmente por Anders Hejlsberg; se anunció en el año 2000 junto con la plataforma .NET.
- .NET es la plataforma que ejecuta tu código C# y le da bibliotecas, herramientas y portabilidad: el SDK sirve para desarrollar, el runtime para ejecutar, y las versiones modernas (.NET 5 en adelante) son gratuitas, open source y multiplataforma (Windows, macOS, Linux).
- El tipado fuerte hace que el compilador atrape errores antes de ejecutar el programa, lo que reduce bugs en producción y hace el código más mantenible en proyectos grandes y equipos profesionales.
- C# brilla en videojuegos (Unity), aplicaciones empresariales, backend web (ASP.NET Core), apps de escritorio y la nube; un solo lenguaje te abre muchísimas puertas laborales.
- Tu código C# se compila a un lenguaje intermedio (IL) y luego el CLR lo traduce a código máquina justo en el momento de ejecutarse: eso es la compilación JIT (Just-In-Time), que da portabilidad y optimización automática.
- Compilar sin errores no significa que el programa esté bien: el compilador revisa sintaxis y tipos, pero los errores de lógica (el programa hace algo incorrecto) solo los detectas tú probando.
- Instalaste el .NET SDK (verificado con `"dotnet --version"`) y Visual Studio Code con la extensión oficial de C#, que te da colores, autocompletado (IntelliSense) y detección de errores en vivo.
- El flujo profesional es: `"dotnet new console -n NombreProyecto"` para crear, editar Program.cs en VS Code (abriendo la carpeta del proyecto), y `"dotnet run"` para compilar y ejecutar.
- El método Main (`"static void Main(string[] args)"`) es el punto de entrada: ahí empieza la ejecución; `"using"` importa namespaces, `"Console.WriteLine"` imprime con salto de línea, y cada instrucción termina con punto y coma.
- Los top-level statements permiten escribir programas de consola en una sola línea, pero debes conocer también la estructura clásica (namespace, clase, Main) porque es la que verás en código existente y entrevistas.
- Los comentarios (`//` y `/* */`) son para humanos, no para la máquina: explica el porqué, no el qué, y mantenlos actualizados.

- Los errores clásicos del principiante (punto y coma olvidado, mayúsculas/minúsculas, comillas simples vs dobles, no guardar antes de ejecutar) se corrigen con práctica y leyendo con calma los mensajes del compilador.

UNIDAD 2: Variables, operadores y entrada de datos

En la unidad anterior instalaste tus herramientas y corriste tu primer programa. Pero seamos honestos: un programa que siempre imprime lo mismo es como un loro que solo sabe una frase: divertido un rato, inútil después. Los programas de verdad hacen algo mucho más interesante: reciben datos, los guardan, los transforman y muestran resultados. Una calculadora recibe números y devuelve la suma; un videojuego guarda tu puntuación y la actualiza; un sistema bancario registra depósitos y calcula saldos. Todo eso (guardar, transformar y mostrar datos) es lo que vas a aprender en esta unidad, y es el corazón de la programación: sin variables no hay memoria, sin operadores no hay cálculo, y sin entrada de datos no hay interacción.

Esta unidad cubre tres bloques que trabajan juntos. Primero, las variables y los tipos de datos: aprenderás qué es una variable (una caja con etiqueta donde guardas un valor), los tipos fundamentales de C# (int para enteros, double y decimal para decimales, string para texto, char para un carácter, bool para verdadero/falso), la palabra mágica "var" (que le pide al compilador que deduzca el tipo), las constantes con "const" (valores que no cambian), y una distinción profunda pero esencial: tipos por valor versus tipos por referencia (cómo se copian los datos en memoria). Segundo, los operadores: los aritméticos (suma, resta, multiplicación, división, módulo), los de comparación (igual, distinto, mayor, menor), los lógicos (Y, O, NO), los de incremento y decremento (++ y --), y las reglas de precedencia (qué se calcula primero cuando hay varias operaciones). Tercero, la entrada de datos: cómo leer lo que el usuario escribe con Console.ReadLine, cómo convertir ese texto a números con int.Parse, Convert y el seguro int.TryParse, y cómo construir mensajes elegantes con interpolación de cadenas (\$"Hola {nombre}").

¿Por qué importa esto en la vida real? Porque todo programa útil es, en el fondo, una máquina de transformar datos. Cuando pides un préstamo en el banco, un programa toma tu sueldo (entrada), lo guarda en variables, aplica operadores (calcula intereses, compara montos) y te muestra si calificas (salida). Cuando juegas en tu celular, el juego guarda tu vida, tus monedas y tu nivel en variables, y usa operadores para sumarte puntos o restarte daño. Hasta la inteligencia artificial, en su nivel más básico, son variables y operaciones matemáticas a escala masiva. Dominar variables, operadores y entrada de datos no es "el tema fácil del inicio": es el 80 por ciento de lo que harás todos los días como programador. Las estructuras avanzadas (clases, listas, bases de datos) son solo formas más organizadas de guardar y transformar datos.

Al terminar esta unidad podrás escribir programas interactivos de verdad: que le pregunten al usuario su nombre y su edad, que calculen promedios, áreas, descuentos e intereses, que tomen decisiones simples con comparaciones y que manejen errores comunes de conversión sin romperse. También habrás desarrollado el instinto más valioso del programador principiante: pensar en tipos. Cada vez que guardes un dato te preguntarás "¿qué tipo es? ¿entero o decimal? ¿puede cambiar o es constante?", y esa pregunta te va a evitar una familia entera de errores.

Un aviso de colega: esta unidad tiene más "trampas" que la anterior. La división entera ($7 / 2 = 3$, no 3.5), la diferencia entre Parse y TryParse, el desbordamiento de tipos, la precedencia de operadores... son los errores clásicos que todo principiante comete y que los profesionales ya ni piensan porque los tienen interiorizados. Te voy a mostrar cada trampa con ejemplos, te voy a explicar por qué ocurre y cómo evitarla. Presta atención especial a las secciones de errores comunes: valen oro.

Explicación detallada

Variables: cajas con etiqueta para guardar datos

Una variable es un espacio de memoria con un nombre, donde guardas un valor que tu programa puede usar y modificar. La analogía clásica (y la mejor) es la de las cajas etiquetadas: imagina una bodega con cajas; cada caja tiene una etiqueta ("edad", "nombre", "precio") y dentro hay un valor. Cuando tu programa necesita la edad del usuario, va a la caja etiquetada "edad", abre la tapa y lee el valor. Cuando el usuario cumple años, el programa abre esa caja, saca el valor viejo y mete el nuevo.

En C#, declarar una variable tiene una sintaxis precisa:

```
int edad = 25;
```

Vamos por partes: "int" es el tipo de dato (número entero), "edad" es el nombre de la variable, "=" es el operador de asignación (mete el valor 25 dentro de la caja), "25" es el valor inicial, y ";" cierra la instrucción. Puedes declarar y asignar en dos pasos si quieres:

```
int edad;        // declaración: creo la caja vacía
edad = 25;       // asignación: meto el valor en la caja
```

Ojo con una sutileza importante: en C#, una variable local (declarada dentro de un método) NO tiene valor por defecto; si intentas usarla antes de asignarle algo, el compilador te detiene

con un error ("use of unassigned local variable"). Esto es el tipado fuerte cuidándote: te obliga a inicializar antes de usar. Acostúmbrate a declarar e inicializar en la misma línea siempre que puedas.

Los nombres de variables tienen reglas y convenciones. Reglas (el compilador las exige): deben empezar con letra o guion bajo, no pueden ser palabras reservadas del lenguaje (como "int", "class", "static"), y no pueden contener espacios ni la mayoría de símbolos. Convenciones (los profesionales las siguen): usa camelCase para variables locales (primera palabra en minúscula, las siguientes con mayúscula inicial: "edadDelUsuario", "precioTotal", "nombreCompleto"), y que el nombre sea descriptivo. "x" no dice nada; "temperaturaEnCelsius" lo dice todo. Tu yo del futuro (y tus compañeros) te lo agradecerán cuando lean tu código dentro de seis meses. Evita las abreviaturas crípticas ("cant", "imp", "aux"): ahorras dos segundos al escribir y pierdes minutos cada vez que alguien (incluido tú) tiene que adivinar qué significan.

Los tipos de datos fundamentales

C# es de tipado fuerte: cada variable tiene un tipo, y el tipo define qué valores puede guardar y qué operaciones permite. Elegir el tipo correcto es una de las decisiones más frecuentes del programador. Vamos con los fundamentales, de los números al texto.

int: números enteros. El tipo "int" (de integer) guarda números enteros de 32 bits, desde -2,147,483,648 hasta 2,147,483,647. Sirve para edades, cantidades, contadores, años, días del mes: todo lo que no tenga decimales. Ejemplo: "int edad = 30;". Es el tipo numérico por defecto: si escribes un número sin decimales en tu código, C# lo trata como int.

Existe toda una familia de enteros por si necesitas otros tamaños: "byte" (0 a 255, útil para datos pequeños), "short" (enteros cortos), "long" (enteros gigantes, hasta 9 trillones aproximadamente). Para el 95 por ciento de tus programas, "int" es suficiente; usa "long" cuando manejes cantidades enormes (como milisegundos desde 1970 o identificadores masivos).

double: números con decimales (punto flotante). El tipo "double" guarda números con parte decimal usando 64 bits, como 3.14159 o 19.99. Es el tipo decimal por defecto: si escribes "19.99" en tu código, C# lo trata como double. Sirve para mediciones, promedios, temperaturas, porcentajes: todo lo aproximado. Pero atención: double representa los decimales en binario, así que algunos valores (como 0.1) no se representan con exactitud total y pueden aparecer sorpresas como 0.30000000000000004 al sumar 0.1 + 0.2. Para la ciencia y los gráficos esto no importa; para dinero, sí importa muchísimo (ver decimal abajo).

float: el hermano menor de double (32 bits, menos precisión). Se usa cuando necesitas ahorrar memoria con muchísimos números (gráficos 3D, por ejemplo). Para declararlo, el valor lleva "f" al final: "float pi = 3.14f;". En este libro casi siempre usaremos double.

decimal: decimales exactos, el tipo del dinero. El tipo "decimal" (128 bits) representa los decimales en base 10, no en binario, así que $0.1 + 0.2 = 0.3$ exactamente, sin sorpresas. Es el tipo OBLIGATORIO para dinero, precios, impuestos, saldos bancarios: cualquier cálculo donde un centavo de error sea inaceptable. Se declara con "m" al final (de money): "decimal precio = 19.99m;". Es más lento que double y usa más memoria, pero en finanzas la exactitud manda. Buena práctica profesional que te va a ahorrar problemas reales: dinero siempre en decimal, nunca en double ni float.

string: cadenas de texto. El tipo "string" guarda texto: nombres, mensajes, direcciones, cualquier secuencia de caracteres. Se escribe entre comillas dobles: "string nombre = "Ana";". Los strings tienen superpoderes que iremos descubriendo (longitud, mayúsculas, búsquedas, interpolación), pero por ahora quédate con que es el tipo para texto. Un string puede estar vacío ("") y eso es distinto de no tener valor (null, que veremos más adelante).

char: un solo carácter. El tipo "char" guarda UN solo carácter entre comillas simples: "char inicial = 'A';". Sirve para letras individuales, símbolos, respuestas de sí/no ('s'/'n'). No lo confundas con string: "char" es un carácter, "string" es una secuencia. "char letra = "A";" (con comillas dobles) es un error de compilación.

bool: verdadero o falso. El tipo "bool" (booleano, en honor al matemático George Boole) solo puede valer "true" o "false". Parece humilde, pero es el tipo de las decisiones: todo "if" que verás en la próxima unidad evalúa un bool. Ejemplos: "bool esMayorDeEdad = true;", "bool tieneDescuento = false;". Los operadores de comparación (>, <, ==) producen bools, y los operadores lógicos (&&, ||, !) los combinan. Piensa en el bool como un interruptor: prendido o apagado, sin medias tintas.

var: deja que el compilador deduzca el tipo

C# te permite declarar variables con "var" en lugar del tipo explícito, y el compilador deduce el tipo a partir del valor inicial:

```
var edad = 25;           // el compilador deduce: int
var precio = 19.99;      // deduce: double
var nombre = "Ana";      // deduce: string
var esValido = true;     // deduce: bool
```


Importante: "var" NO significa "sin tipo" ni "tipo dinámico". La variable SIGUE siendo fuertemente tipada; simplemente no escribiste el tipo porque era obvio. "var edad = 25;" crea un int de verdad, y si después intentas "edad = "hola";" el compilador te detendrá. Es azúcar sintáctica: menos escritura, misma seguridad.

¿Deberías usar var siempre? La comunidad está dividida, pero hay consenso práctico: usa "var" cuando el tipo sea obvio por el valor ("var nombre = "Ana";" nadie duda que es string), y usa el tipo explícito cuando aclare la intención ("int totalSegundos = 3600;" deja claro que trabajas con enteros, o "decimal precio = 19.99m;" deja claro que es dinero). En este libro usaremos tipos explícitos en los temas de aprendizaje (para que veas el tipo) y "var" ocasionalmente. Regla de oro: el código debe leerse fácil; si "var" oscurece qué tipo es, escribe el tipo.

Restricciones de var: siempre requiere valor inicial ("var x;" es error, porque no hay nada de donde deducir), solo sirve para variables locales (no para campos de clase ni parámetros), y una vez deducido el tipo no cambia.

const: valores que no cambian

A veces necesitas un valor fijo que tu programa usa pero nunca modifica: el IVA (16 por ciento en México), los días de la semana (7), el nombre de tu aplicación. Para eso está "const" (constante): declaras el valor una vez y el compilador te impide cambiarlo después.

```
const double IVA = 0.16;
const int DIAS_SEMANA = 7;
const string NOMBRE_APP = "Mi Calculadora";
```

Convención: las constantes se escriben en MAYÚSCULAS_CON_GUIONES_BAJOS (aunque el compilador no lo exige, los profesionales lo hacen y así se reconocen a simple vista). Si intentas reasignar una constante ("IVA = 0.20;"), obtienes un error de compilación. Esto es una red de seguridad: evita que por accidente cambies un valor que debía ser fijo. Además, las constantes hacen tu código más legible: "precio * IVA" se entiende mejor que "precio * 0.16" (¿qué es 0.16? ¿por qué? El nombre lo explica). A esos números misteriosos regados por el código se les llama "números mágicos", y reemplazarlos por constantes con nombre es una de las primeras marcas de un programador profesional.

Diferencia rápida: "const" es para valores conocidos en tiempo de compilación y que nunca cambian; existe también "readonly" para valores que se fijan al crear un objeto y ya no cambian (lo verás con clases más adelante). Por ahora, const es tu herramienta.

Tipos por valor vs tipos por referencia: la distinción profunda

Aquí viene un concepto que separa a quien "usa" C# de quien lo "entiende": cómo se almacenan y copian los datos en memoria. Hay dos familias:

Tipos por valor (value types): int, double, decimal, char, bool, float y las estructuras (struct). Cuando asignas una variable de tipo por valor a otra, se COPIA el valor. Cada variable tiene su propia caja independiente con su propia copia.

```
int a = 10;
int b = a;    // se copia el valor 10 a la caja de b
b = 20;       // cambio la caja de b...
// a sigue valiendo 10: son cajas independientes
```

Analogía: es como fotocopiar un documento. Le das la fotocopia a tu amigo; si él escribe encima de SU copia, tu original no se altera. Cada uno tiene su papel.

Tipos por referencia (reference types): string, arreglos, clases, objetos en general. La variable no guarda el dato directamente: guarda una REFERENCIA (una dirección, un apuntador) al lugar de memoria donde vive el dato. Cuando asignas una variable de referencia a otra, se copia la REFERENCIA, no el dato: ambas variables apuntan al MISMO objeto.

Analogía: es como darle a tu amigo la DIRECCIÓN de tu casa en lugar de una fotocopia. Si él va a esa dirección y pinta la puerta de rojo, cuando tú llegues verás la puerta roja: ambos apuntaban a la misma casa.

```
int[] numeros1 = { 1, 2, 3 };
int[] numeros2 = numeros1; // copia la referencia: apuntan al mismo arreglo
numeros2[0] = 99;
// numeros1[0] ahora también vale 99: ¡es el mismo arreglo!
```

(Los arreglos los verás en detalle en su unidad; por ahora importa el principio.)

Caso especial: string. Técnicamente string es un tipo por referencia, pero se comporta como por valor en la práctica porque los strings son INMUTABLES: no puedes modificar un string existente, solo crear uno nuevo. Cuando haces "nombre = nombre + " Pérez";", en realidad se crea un string nuevo y la variable apunta al nuevo. Así que para efectos prácticos, con strings puedes pensar en copias independientes sin miedo.

¿Por qué te cuento esto tan temprano? Porque este concepto explica una familia entera de bugs "misteriosos" ("¿por qué cambió mi variable si yo no la toqué?") y porque es pregunta clásica de entrevistas de trabajo. No necesitas dominarlo hoy al cien: necesitas saber que

existe, y volverás a él con más herramientas en la unidad de clases y objetos. Por ahora, regla práctica: con `int`, `double`, `decimal`, `char` y `bool` (tipos por valor), las asignaciones copian y cada variable vive su vida; con objetos (arreglos, clases), las asignaciones comparten.

Operadores aritméticos: la calculadora del programador

Los operadores aritméticos hacen matemáticas con tus variables. C# tiene los que esperas y uno extra muy útil:

```
int suma = 10 + 5;           // 15: suma
int resta = 10 - 5;          // 5: resta
int multiplicacion = 10 * 5; // 50: multiplicación (asterisco, no "x")
int division = 10 / 5;       // 2: división
int modulo = 10 % 3;         // 1: módulo (el residuo de la división)
```

El operador módulo (%) merece atención especial porque muchos principiantes no lo conocen y es utilísimo: te da el RESIDUO de una división entera. $10 \% 3 = 1$ porque 10 entre 3 cabe 3 veces (9) y sobra 1. ¿Para qué sirve en la vida real? Para saber si un número es par o impar ("`numero % 2 == 0`" significa "es par"), para ciclos que se repiten cada N elementos, para repartir turnos, para validar dígitos. Cuando necesites "cada cuántos" o "sobra algo", piensa en módulo.

LA TRAMPA CLÁSICA: la división entera. Mira esto con atención, porque es el error número uno de esta unidad:

```
int resultado = 7 / 2; // ¿cuánto vale? ¡3, no 3.5!
```

En C#, si divides dos enteros (`int / int`), el resultado es ENTERO: se trunca la parte decimal, sin redondear. $7 / 2 = 3$ (no 3.5, no 4). Esto no es un bug: es el comportamiento definido del lenguaje, heredado de C. Pero destruye promedios y cálculos si no lo esperas: un programa que calcula el promedio de 7 y 2 como " $(7 + 2) / 2$ " te dará 4 en lugar de 4.5.

¿Cómo obtener el resultado decimal? Asegúrate de que al menos uno de los operandos sea decimal:

```
double resultado1 = 7.0 / 2; // 3.5 (7.0 es double)
double resultado2 = 7 / 2.0; // 3.5
double resultado3 = (double)7 / 2; // 3.5: conversión explícita (casting)
```

Ese "`(double)7`" es un cast (conversión explícita): le dices al compilador "trata este 7 como double". Lo verás más en la sección de conversiones. Regla práctica: si tu cálculo involucra

división y te importan los decimales, trabaja con double o decimal desde el principio. Y si calculas un promedio de enteros, convierte antes de dividir, no después: "(double)(a + b) / 2" está bien; "(double)((a + b) / 2)" está mal (la división entera ya truncó antes del cast).

Otro detalle: la división entre decimales sí da decimales ("7.0 / 2" = 3.5), y dividir entre cero con enteros lanza una excepción (el programa se cae con DivideByZeroException), mientras que con double da Infinito. Manejar divisiones por cero es tu responsabilidad: más adelante aprenderás validaciones; por ahora, sé consciente del riesgo.

También existen los operadores de asignación compuesta, atajos que combinan operación y asignación:

```
int x = 10;
x += 5;    // x = x + 5  -> 15
x -= 3;    // x = x - 3  -> 12
x *= 2;    // x = x * 2  -> 24
x /= 4;    // x = x / 4  -> 6
x %= 4;    // x = x % 4  -> 2
```

Son muy usados para acumuladores ("total += precio;" suma cada precio al total) y contadores. Úsalos: son más cortos y los profesionales los esperan.

Operadores de comparación: preguntas que responden true o false

Los operadores de comparación hacen preguntas sobre valores y responden con un bool (true/false). Son la materia prima de las decisiones (los "if" de la próxima unidad):

```
int edad = 18;
bool esMayor = edad >= 18;    // true: ¿edad es mayor o igual que 18?
bool esMenor = edad < 18;     // false
bool esIgual = edad == 18;    // true: ¿edad es igual a 18?
bool esDistinto = edad != 18; // false: ¿edad es distinto de 18?
```

La lista completa: "==" (igual que), "!=" (distinto de), ">" (mayor que), "<" (menor que), ">=" (mayor o igual), "<=" (menor o igual). LA TRAMPA CLÁSICA número dos: confundir "=" (asignación: mete un valor en la variable) con "==" (comparación: pregunta si son iguales). "if (edad = 18)" asigna 18 a edad y casi siempre es un error lógico; "if (edad == 18)" pregunta. El compilador de C# te ayuda aquí: en muchos contextos donde escribes "=" por accidente dentro de una condición, te avisa. Pero no siempre, así que entrena tu ojo: un igual asigna, dos iguales comparan.

Comparar decimales con "==" tiene un matiz: por la representación binaria de double, dos cálculos que "deberían" dar lo mismo pueden diferir en el último decimal (0.1 + 0.2 == 0.3 da false con double). Para dinero usa decimal (exacto) y para comparaciones aproximadas con double compara la diferencia contra una tolerancia pequeña. Es un detalle avanzado, pero te lo dejo sembrado porque algún día te morderá.

Los strings también se comparan con "==" en C# (a diferencia de Java, donde hay que usar un método): "nombre == "Ana"" funciona y compara el CONTENIDO. Para comparaciones sin importar mayúsculas/minúsculas existen métodos específicos que verás en la unidad de cadenas.

Operadores lógicos: combinar preguntas

Los operadores lógicos combinan valores bool para formar preguntas compuestas:

- "&&" (Y lógico, AND): true solo si AMBOS lados son true. "¿Es mayor de edad Y tiene credencial?" Ambas deben cumplirse.
- "||" (O lógico, OR): true si AL MENOS UNO es true. "¿Tiene tarjeta O efectivo?" Con una basta.
- "!" (NO lógico, NOT): invierte el valor. "!true" es false. "¿NO tiene descuento?".

```
int edad = 20;
bool tieneCredencial = true;
bool puedeVotar = edad >= 18 && tieneCredencial; // true && true = true

bool esFinDeSemana = false;
bool esFestivo = true;
bool descansa = esFinDeSemana || esFestivo; // false || true = true

bool tieneDeuda = true;
bool estaLimpio = !tieneDeuda; // false
```

Tablas de verdad (memorízalas, son la base de toda decisión en programación):

- AND (&&): true && true = true; cualquier otro = false.
- OR (||): false || false = false; cualquier otro = true.
- NOT (!): invierte.

Detalle profesional: "&&" y "||" son de "cortocircuito" (short-circuit): si el lado izquierdo ya decide el resultado, el derecho ni se evalúa. En "false && Algo()", "Algo()" nunca se ejecuta; en "true || Algo()", tampoco. Esto no es solo optimización: se usa como técnica de seguridad.

Ejemplo: "if (texto != null && texto.Length > 0)": si texto es null, la primera condición es false y la segunda (que se caería al intentar leer Length de null) ni se evalúa. Sin cortocircuito, ese código se rompería. Existe también "&" y "|" simples (sin cortocircuito), pero en la práctica casi siempre querrás "&&" y "||".

Incremento, decremento y precedencia

Los operadores "++" (incremento) y "--" (decremento) suman o restan 1 a una variable. Son los favoritos de los ciclos:

```
int contador = 0;
contador++; // contador ahora vale 1 (equivale a contador = contador + 1)
contador++; // vale 2
contador--; // vale 1
```

Sutileza que confunde: la posición importa cuando el operador va DENTRO de una expresión. "++contador" (pre-incremento) primero incrementa y luego usa el valor; "contador++" (post-incremento) primero usa el valor y luego incrementa:

```
int a = 5;
int b = a++; // b = 5 (usa el valor viejo), a queda en 6
int c = ++a; // a sube a 7 primero, c = 7
```

Consejo profesional: evita usar ++/-- dentro de expresiones complejas. Úsalos solos, en su propia línea ("contador++;"), donde no hay ambigüedad. El código que requiere pensar dos veces para entender el orden de evaluación es código que generará bugs. La claridad siempre le gana a la astucia.

Precedencia de operadores: cuando una expresión tiene varias operaciones, C# sigue un orden (como en matemáticas): primero paréntesis, luego multiplicación/división/módulo, luego suma/resta, luego comparaciones, luego lógicos (NOT, AND, OR). Ejemplo:

```
int r = 2 + 3 * 4; // 14, no 20: primero 3*4=12, luego +2
bool b = 5 > 3 && 2 < 4; // true: primero las comparaciones, luego el &&
```

Pero no memorices tablas de precedencia: usa PARÉNTESIS. "int r = 2 + (3 * 4);" dice lo mismo y nadie tiene que pensar. Los paréntesis son gratis y eliminan una categoría entera de errores tontos. Los profesionales los usan generosamente, no por ignorancia sino por claridad: el código se lee una vez y se entiende, en lugar de obligar al lector a recordar reglas.

Desbordamiento (overflow): cuando el número no cabe

Cada tipo numérico tiene un rango. ¿Qué pasa si te pasas? Con "int", el máximo es 2,147,483,647. Si le sumas 1... no da error: SE DESBORDA y se convierte en -2,147,483,648 (el mínimo). Es como el odómetro de un coche viejo que al pasar de 999,999 vuelve a 000,000. Por defecto, C# NO avisa: el cálculo sigue con un valor incorrecto en silencio. Este es uno de los bugs más traicioneros porque el programa no se cae: simplemente calcula mal.

```
int maximo = 2147483647;
maximo++; // ¡ahora vale -2147483648! Desbordamiento silencioso.
```

¿Cómo protegerte? (1) Elige tipos con margen suficiente: si manejas cantidades que pueden crecer mucho, usa "long". (2) Existe el contexto "checked", que convierte el desbordamiento en excepción (el programa avisa en lugar de calcular mal): "checked { maximo++; }" lanzaría OverflowException. (3) Valida entradas: si el usuario escribe un número gigantesco donde esperabas una edad, algo anda mal y tu programa debería decirlo. En la práctica diaria el desbordamiento es raro, pero cuando muerde, muerde en producción y con dinero de por medio: por eso los sistemas financieros usan decimal (rango enorme) y validaciones estrictas.

Entrada de datos: Console.ReadLine

Hasta ahora tus programas hablan pero no escuchan: muestran información pero el usuario no puede responder. "Console.ReadLine()" cambia eso: pausa el programa, espera a que el usuario escriba algo y presione Enter, y devuelve lo escrito como un string.

```
Console.Write("¿Cómo te llamas? ");
string nombre = Console.ReadLine();
Console.WriteLine("Hola, " + nombre + ". ¡Bienvenido!");
```

Línea por línea: "Console.Write" muestra la pregunta SIN saltar de línea (para que el cursor quede justo después, esperando la respuesta en el mismo renglón: detalle de programa real). "Console.ReadLine()" lee la línea que el usuario escribe y la guarda en la variable "nombre". La tercera línea saluda usando concatenación (unir strings con "+").

Punto crucial: ReadLine SIEMPRE devuelve un string, incluso si el usuario escribió un número. Si escribes "25", recibes el TEXTO "25", no el número 25. No puedes hacer matemáticas con texto: "Console.ReadLine() + 5" con entrada "25" daría "255" (pegado de textos), no 30. Por eso necesitas conversiones, que es el siguiente tema. Y ojo: si el usuario solo presiona Enter sin escribir nada, ReadLine devuelve un string vacío (""); si la entrada se cierra (caso raro en consola interactiva), puede devolver null. Un programa robusto valida estas situaciones.

"Console.ReadKey()" es el primo de ReadLine: lee UNA tecla sin esperar Enter (útil para "presiona cualquier tecla para continuar"). Lo menciono porque lo verás en ejemplos; en este libro nos centraremos en ReadLine.

Conversiones: de texto a número (Parse, Convert, TryParse)

Tienes el texto "25" y necesitas el número 25. C# te da tres caminos, del más frágil al más robusto. Entender cuándo usar cada uno es marca de profesional.

Opción 1: int.Parse. Convierte un string a entero. Si el texto es un número válido, funciona perfecto; si NO lo es (letras, vacío, decimales con punto), LANZA UNA EXCEPCIÓN y tu programa se cae.

```
string texto = "25";
int numero = int.Parse(texto); // numero = 25

string malo = "hola";
int error = int.Parse(malo); // ¡EXCEPCIÓN! El programa se cae aquí.
```

"Parse" es directo y legible, y está bien usarlo cuando estás SEGURO de que el texto es válido (por ejemplo, un dato que viene de tu propia base de datos, no de un humano). Para entrada de usuario, es una trampa: los humanos escriben "veinticinco", dejan espacios, ponen comas. Nunca confíes en lo que escribe un usuario.

Opción 2: Convert.ToInt32 (y familia). La clase "Convert" tiene métodos para convertir entre tipos: "Convert.ToInt32("25")", "Convert.ToDouble("3.14")", "Convert.ToString(42)", etc. Se comporta parecido a Parse con texto inválido (lanza excepción), con una diferencia notable: si le pasas null, devuelve 0 en lugar de caerse. Es versátil para conversiones generales entre tipos, no solo string a número.

Opción 3: int.TryParse (LA FORMA CORRECTA para entrada de usuario). TryParse intenta convertir y, en lugar de caerse si falla, devuelve un bool: true si pudo, false si no. El número convertido sale por un parámetro "out":


```

Console.Write("Escribe tu edad: ");
string entrada = Console.ReadLine();

if (int.TryParse(entrada, out int edad))
{
    Console.WriteLine("Tienes " + edad + " años.");
}
else
{
    Console.WriteLine("Eso no es un número válido.");
}

```

Si el usuario escribe "30", TryParse devuelve true y "edad" vale 30. Si escribe "treinta", devuelve false, "edad" vale 0, y tu programa muestra el mensaje amable en lugar de explotar. Esa es la diferencia entre un programa de estudiante y uno profesional: el profesional ASUME que la entrada será incorrecta y la maneja con elegancia.

El patrón "out int edad" declara la variable directamente dentro de la llamada (sintaxis moderna de C# 7+): antes había que declararla en una línea separada. TryParse existe para todos los numéricos: double.TryParse, decimal.TryParse, etc. Y hay variantes culturales: "double.TryParse("3,14")" puede fallar o no según la configuración regional (en México el separador decimal es el punto en programación, la coma en otros países). Para este libro, asume punto decimal en el código.

Resumen de decisión: ¿el dato viene de una fuente confiable y controlada? Parse está bien. ¿Viene de un humano? TryParse, siempre. ¿Necesitas convertir entre tipos variados (no solo texto)? Convert. Grábate esto y evitarás la categoría de bugs más vergonzosa: el programa que se cae porque el usuario escribió algo inesperado.

También existe el casting (conversión explícita entre numéricos) que vimos antes: "(double)7" o "int parteEntera = (int)3.99;" (trunca a 3). El cast entre numéricos es para cuando YA tienes un número y quieres otro tipo; Parse/TryParse/Convert son para cuando tienes TEXTO. No los confundas: son herramientas para problemas distintos.

Interpolación de cadenas: mensajes elegantes

Unir textos con "+" funciona pero se vuelve ilegible rápido: ""Hola " + nombre + ", tienes " + edad + " años."" es un rompecabezas de comillas y más. La interpolación de cadenas (desde C# 6) es la forma moderna y legible: pones "\$" antes del string y metes las variables entre llaves:

```
string nombre = "Ana";
int edad = 30;
Console.WriteLine($"Hola {nombre}, tienes {edad} años.");
// Salida: Hola Ana, tienes 30 años.
```

Dentro de las llaves puedes poner expresiones, no solo variables: `$"El doble de {edad} es {edad * 2}."` Y puedes formatear números: `$"Precio: {precio:C}"` muestra el valor como moneda según tu región (\$19.99); `$"Pi: {pi:F2}"` muestra solo 2 decimales (3.14); `$"Porcentaje: {tasa:P1}"` lo muestra como porcentaje (16.0%). Estos formatos ("C" moneda, "F2" dos decimales, "P" porcentaje, "D" entero con ceros) son estándar profesional para reportes.

La interpolación también funciona con "verbatim strings" combinados (`$@"..."`) y es la forma que usaremos en todo el libro. Regla simple: si tu mensaje tiene más de una variable, usa interpolación; el "+" déjalo para uniones triviales.

Errores comunes de esta unidad (la galería de trampas)

1. División entera inesperada. `"int promedio = (8 + 9) / 2;"` da 8, no 8.5. Solución: usa `double/decimal` o convierte antes de dividir. Cada vez que dividas, pregúntate: "¿los operandos son enteros? ¿me importa el decimal?"
2. Parse con entrada inválida. `"int.Parse(Console.ReadLine())"` se cae si el usuario escribe letras. Solución: `TryParse` para todo lo que venga de humanos.
3. Comparar con `"="` en lugar de `"=="`. `"if (x = 5)"` asigna en vez de comparar. Solución: dos iguales para comparar; lee tus condiciones en voz alta ("¿x es igual a 5?").
4. `double` para dinero. `"double total = 0.1 + 0.2;"` puede dar 0.30000000000000004. Solución: `decimal` con sufijo `"m"` para todo lo financiero.
5. Olvidar que `ReadLine` devuelve `string`. Intentar `"string edad = Console.ReadLine(); int doble = edad * 2;"` no compila (no se puede multiplicar texto). Solución: convierte primero con `TryParse`.
6. Concatenar números como texto sin querer. Con entrada `"25"` (string), `"entrada + entrada"` da `"2525"`. Solución: convierte a número ANTES de operar.
7. Variables sin inicializar. `"int x; Console.WriteLine(x);"` no compila. Solución: inicializa al declarar.
8. Nombres poco descriptivos. `"int a1, b2;"` funciona pero es ilegible. Solución: `"int calificacionParcial, calificacionFinal;"`.

9. Desbordamiento silencioso. Sumar más allá del máximo de int sin darte cuenta. Solución: usa long si hay duda, valida entradas.
10. Confundir char con string. "char c = "A";" no compila (comillas dobles = string). Solución: comillas simples para char, dobles para string.

Buenas prácticas profesionales

- Nombres descriptivos en camelCase para variables ("precioConDescuento"), MAYÚSCULAS para constantes ("IVA"). Tu código debe leerse como una historia.
- Inicializa al declarar. "int contador = 0;" es mejor que declarar vacío y asignar después.
- decimal para dinero, double para ciencia/mediciones, int para conteos. El tipo comunica intención.
- const para valores fijos (tasas, límites, nombres). Nada de números mágicos regados.
- TryParse para entrada de usuario, siempre. Un programa que se cae por un dato mal escrito es un programa sin terminar.
- Interpolación ("...") para mensajes con variables. Legibilidad ante todo.
- Paréntesis generosos en expresiones con varios operadores. La claridad le gana a la memorización de precedencias.
- Valida rangos, no solo tipos: que TryParse diga "es un número" no significa que "sea una edad válida" (¿-5 años? ¿1000?). Las validaciones de negocio son tu responsabilidad.
- Un concepto por variable: si una variable cambia de significado a mitad del programa, necesitas dos variables. Las variables reutilizadas para cosas distintas son fuente de confusión.

Ámbito de variables: dónde vive y dónde muere cada variable

No basta con saber declarar variables: hay que saber DÓNDE existen. El "ámbito" (scope) de una variable es la región del código donde esa variable es visible y usable. En C#, el ámbito lo definen las llaves "{}": una variable existe desde la línea donde se declara hasta la llave de cierre del bloque que la contiene.

```
int edad = 30; // ámbito: todo el método

if (edad >= 18)
{
    string mensaje = "Es mayor de edad"; // ámbito: solo dentro de este if
    Console.WriteLine(mensaje); // OK: estoy dentro del bloque
}
// Console.WriteLine(mensaje); // ERROR: mensaje ya no existe aquí
```

La variable "mensaje" nace dentro de las llaves del if y muere al cerrarse esas llaves. Intentar usarla fuera es un error de compilación ("the name 'mensaje' does not exist in the current context"). Esto no es capricho del lenguaje: es una característica de seguridad. Los bloques son como cuartos con puerta: lo que declaras dentro de un cuarto se queda en ese cuarto. Así evitas choques de nombres y el programa libera memoria automáticamente cuando el bloque termina.

Tres reglas prácticas que te evitarán dolores de cabeza:

1. Declara las variables lo más cerca posible de donde las usas, en el bloque más pequeño que las necesite. Si un contador solo se usa dentro de un if, decláralo dentro del if. Esto hace el código más fácil de entender: el lector ve la variable y su uso juntos.
2. No puedes declarar dos variables con el mismo nombre en el mismo ámbito ("int x = 1; int x = 2;" es error), pero SÍ puedes reutilizar el nombre en bloques distintos y anidados (aunque es confuso y mejor evitarlo).
3. Una variable declarada en un bloque externo es visible en los bloques internos (el cuarto grande ve hacia los cuartos chicos, pero no al revés). "edad" declarada en el método se puede usar dentro del if, como viste en el ejemplo.

El concepto de ámbito será crucial cuando llegues a métodos (cada método tiene su propio ámbito) y a clases. Por ahora quédate con la imagen mental: las llaves son fronteras, y las variables no cruzan fronteras hacia afuera.

Conversiones entre tipos: implícitas, explícitas y la jerarquía

Ya viste Parse/TryParse (texto a número). Ahora veamos las conversiones ENTRE tipos numéricos, que tienen sus propias reglas. C# distingue dos clases:

Conversiones implícitas (automáticas, seguras): cuando el tipo destino puede contener CUALQUIER valor del tipo origen sin perder información, C# convierte solo, sin que pidas permiso. La jerarquía va de menor a mayor capacidad: int -> long -> float -> double -> decimal (con matices). Ejemplos:

```
int entero = 100;
long grande = entero;      // implícita: todo int cabe en long
double real = entero;      // implícita: todo int cabe en double
double d = 5;              // el 5 (int) se convierte solo a double
```

Conversiones explícitas (cast, manuales, potencialmente con pérdida): cuando el tipo destino es MÁS PEQUEÑO o la conversión puede perder información, C# te exige pedirla explícitamente con paréntesis, como diciendo "sé lo que hago":

```
double pi = 3.99;
int aproximado = (int)pi; // 3: se TRUNCA la parte decimal (no redondea)

long poblacion = 90000000000L;
int recortado = (int)poblacion; // ¡peligro! el valor no cabe: resultado incorrecto
```

El cast de double a int TRUNCA (corta los decimales), no redondea: (int)3.99 = 3. Si quieres redondear, usa "Math.Round(3.99)" que da 4. Y el cast de long a int cuando el valor no cabe produce desbordamiento silencioso: otro caso donde el programa no se cae pero calcula mal.

La clase Convert también hace estas conversiones: "Convert.ToInt32(3.99)" REDONDEA (da 4), a diferencia del cast que trunca. Detalle sutil pero importante: (int)3.99 = 3, Convert.ToInt32(3.99) = 4. Cuando conviertas decimales a enteros, decide conscientemente: ¿quiero truncar o redondear? Los bugs financieros nacen de no decidir esto.

¿Y de número a texto? Fácil: todo tiene el método "ToString()": "int edad = 30; string texto = edad.ToString();". Y con formato: "precio.ToString("C")" da formato de moneda. La interpolación que ya conoces ("\$"...") usa ToString por dentro.

Tabla mental de decisiones para conversiones:

- Texto -> número (entrada de usuario): TryParse.
- Texto -> número (fuente confiable): Parse.
- Número pequeño -> número grande: asignación directa (implícita).

- Número grande/decimal -> número pequeño/entero: `cast (int)`, decidiendo truncar vs redondear (`Convert.ToInt32` o `Math.Round`).
- Número -> texto: `ToString()` o interpolación.
- Cualquier cosa -> texto para mostrar: interpolación `$"..."`.

Un error común: intentar convertir con `cast` lo que requiere `Parse`. `"(int)"hola"` no compila: el `cast` no convierte texto, solo cambia entre tipos numéricos compatibles. Texto a número es trabajo de `Parse/TryParse/Convert`, punto.

El operador ternario y el manejo de nulos: atajos elegantes

Hay dos herramientas compactas que los programadores C# usan a diario y que vale la pena conocer desde ahora: el operador ternario y los operadores de nulos. El ternario ya apareció en una solución de práctica; aquí lo explicamos en serio.

El operador ternario `"? :"` es un `if-else` compacto para ASIGNAR un valor según una condición. Su forma es: `condición ? valorSiTrue : valorSiFalse`.

```
int edad = 20;
string categoria = edad >= 18 ? "Adulto" : "Menor";
// Si edad >= 18 es true, categoria = "Adulto"; si no, "Menor".

int puntos = 85;
string resultado = puntos >= 60 ? "Aprobado" : "Reprobado";
```

Léelo como una pregunta: `"¿edad >= 18? Entonces 'Adulto', si no 'Menor'"`. Es perfecto para asignaciones simples basadas en una condición. Pero tiene una regla de estilo estricta entre profesionales: ÚSALO SOLO PARA CASOS SIMPLES. En cuanto la expresión crece, se vuelve ilegible:

```
// BIEN: simple y claro
string saludo = esDeDia ? "Buenos días" : "Buenas noches";

// MAL: ternario anidado, un acertijo
string tipo = edad < 13 ? "Niño" : edad < 18 ? "Adolescente" : edad < 65 ? "Adulto"
: "Mayor";
```

Ese segundo ejemplo compila, pero nadie lo entiende a la primera. Para lógica con más de dos ramas, usa `if-else` (próxima unidad). El ternario es un bisturí, no un machete.

Ahora, los nulos. En C#, los tipos por referencia (como string) pueden valer "null", que significa "sin valor, vacío, nada". Es distinto de un string vacío (""): "" es una caja vacía, null es la ausencia de caja. Esta distinción causa el error más famoso de .NET: `NullReferenceException` ("referencia nula"), que ocurre cuando intentas usar algo que es null, como pedirle la longitud a un string que no existe:

```
string nombre = null;
int largo = nombre.Length; // ¡EXCEPCIÓN! No hay string del cual medir nada.
```

Para defenderte, C# moderno te da dos operadores:

1. `"?."` (acceso condicional, null-conditional): `nombre?.Length` significa "si nombre no es null, dame su Length; si es null, el resultado es null en lugar de explotar". Es como tocar la puerta antes de entrar: si no hay nadie, no pasa nada malo.
2. `"??"` (fusión de nulos, null-coalescing): `nombre ?? "Sin nombre"` significa "si nombre no es null, úsalo; si es null, usa 'Sin nombre'". Es el valor de respaldo:

```
Console.Write("Escribe tu nombre (o Enter para anónimo): ");
string entrada = Console.ReadLine();
string nombreFinal = string.IsNullOrEmpty(entrada) ? "Anónimo" : entrada;
Console.WriteLine($"Hola, {nombreFinal}.");
```

Aquí combiné ternario con `"string.IsNullOrEmpty"` (método que detecta null O vacío, utilísimo para validar entradas). Si el usuario solo presiona Enter, el nombre es "Anónimo" en lugar de un saludo con hueco.

Desde C# 8 existe además el contexto de "tipos de referencia anulables" (nullable reference types), donde el compilador te avisa si olvidas manejar posibles nulos en strings y objetos. Es un tema avanzado para su unidad, pero la filosofía ya la tienes: en C# moderno, los nulos se manejan de forma explícita y defensiva, no se ignoran. Cada vez que recibas un string de `ReadLine` o de cualquier fuente externa, pregúntate: "¿y si viene null o vacío?" Esa pregunta te pone por delante del 90 por ciento de los bugs de principiantes.

Ejemplos paso a paso

Vamos a la práctica guiada. Cada ejemplo es un programa completo que puedes crear con `"dotnet new console -n NombreEjemplo"`, escribir a mano y ejecutar. Después de cada bloque explico línea por línea lo relevante. Escribe, ejecuta, experimenta: cambia valores y observa.

Ejemplo 1: Declarando variables de cada tipo

El tour completo por los tipos fundamentales en un solo programa. Úsalo como referencia rápida cuando dudes qué tipo elegir.

```
// Declaración de variables de los tipos fundamentales
int edad = 30; // entero: sin decimales
long poblacion = 128000000L; // entero grande: sufijo L opcional
double temperatura = 36.6; // decimal aproximado (punto flotante)
decimal precio = 19.99m; // decimal exacto: sufijo m (dinero)
float velocidad = 88.5f; // flotante de 32 bits: sufijo f
char inicial = 'E'; // un solo carácter: comillas simples
string nombre = "Esteban"; // texto: comillas dobles
bool esEstudiante = true; // booleano: true o false
const double IVA = 0.16; // constante: valor fijo

Console.WriteLine($"Nombre: {nombre} (inicial {inicial})");
Console.WriteLine($"Edad: {edad}, Estudiante: {esEstudiante}");
Console.WriteLine($"Temperatura: {temperatura}, Velocidad: {velocidad}");
Console.WriteLine($"Precio sin IVA: {precio:C}, IVA: {IVA:P0}");
Console.WriteLine($"Población: {poblacion}");
```

Explicación: cada línea declara una variable con su tipo explícito y un valor inicial (buena práctica: declarar e inicializar juntas). Fíjate en los sufijos: "m" para decimal (dinero), "f" para float, "L" para long (opcional pero aclara). "char" usa comillas simples, "string" dobles. "const double IVA" es la constante con nombre descriptivo en mayúsculas. En las salidas uso interpolación con formatos: "{precio:C}" muestra moneda, "{IVA:P0}" muestra porcentaje sin decimales. Ejecuta y observa cómo cada tipo se muestra: esa intuición visual te servirá para detectar errores de tipo en el futuro.

Ejemplo 2: Operadores aritméticos y la trampa de la división entera

Este ejemplo demuestra en vivo la diferencia entre división entera y decimal, para que la trampa te muerda aquí (en práctica) y no en un examen o trabajo.


```

int a = 7;
int b = 2;

int divisionEntera = a / b;
Console.WriteLine($"7 / 2 como enteros = {divisionEntera}"); // 3, ¡no 3.5!

double divisionDecimal = (double)a / b;
Console.WriteLine($"7 / 2 como decimal = {divisionDecimal}"); // 3.5

int residuo = a % b;
Console.WriteLine($"Residuo de 7 / 2 = {residuo}"); // 1

// Promedio: la forma correcta y la incorrecta
int cal1 = 8, cal2 = 9, cal3 = 7;
double promedioMal = (cal1 + cal2 + cal3) / 3;
double promedioBien = (cal1 + cal2 + cal3) / 3.0;
Console.WriteLine($"Promedio mal: {promedioMal}"); // 8 (¡truncado!)
Console.WriteLine($"Promedio bien: {promedioBien}"); // 8.0

// Operadores de asignación compuesta
int total = 0;
total += 100; // total = 100
total += 50; // total = 150
total *= 2; // total = 300
Console.WriteLine($"Total acumulado: {total}");

```

Explicación línea por línea: "a / b" con ambos int da 3 (trunca el .5): la trampa en acción. "(double)a / b" convierte "a" a double ANTES de dividir, así el resultado es 3.5: fíjate que el cast va sobre el operando, no sobre el resultado. "a % b" da el residuo 1. En los promedios: "promedioMal" divide enteros primero (24/3=8) y luego guarda en double (8.0): el daño ya está hecho; "promedioBien" divide entre 3.0 (double), así la división es decimal desde el inicio. La lección de oro: convierte ANTES de operar, no después. Finalmente, "+=", "*=" acumulan de forma compacta: "total += 100" suma 100 al valor actual.

Ejemplo 3: Comparaciones y lógica - ¿puede entrar al antro?

Un ejemplo con sabor a vida real: un programa que decide si alguien puede entrar a un lugar según edad y credencial, usando comparaciones y operadores lógicos.

```

Console.Write("Edad: ");
int edad = int.Parse(Console.ReadLine()); // en programa real usaríamos TryParse

Console.Write("¿Trae credencial? (s/n): ");
string respuesta = Console.ReadLine();
bool tieneCredencial = respuesta == "s";

bool esMayor = edad >= 18;
bool puedeEntrar = esMayor && tieneCredencial;
bool necesitaPermiso = !esMayor && edad >= 16;

Console.WriteLine($"Es mayor de edad: {esMayor}");
Console.WriteLine($"Puede entrar: {puedeEntrar}");
Console.WriteLine($"Necesita permiso de sus padres: {necesitaPermiso}");

```

Explicación: leemos la edad y la convertimos con Parse (nota el comentario honesto: en producción sería TryParse; aquí lo simplifiqué para enfocarnos en la lógica). "respuesta == "s"" compara strings y produce un bool: si el usuario escribió "s", tieneCredencial es true. "edad >= 18" es la comparación clásica. "esMayor && tieneCredencial": AND, ambas deben ser true para entrar. "!esMayor && edad >= 16": NOT invierte esMayor (es menor), AND con tener al menos 16: necesita permiso. Observa el estilo: guardé cada condición en una variable bool con nombre descriptivo ("puedeEntrar", "necesitaPermiso") en lugar de meter la expresión cruda en el WriteLine. Así el código se lee como español: "puede entrar = es mayor Y tiene credencial". Esa es la diferencia entre código que funciona y código profesional.

Ejemplo 4: Entrada de datos robusta con TryParse - calculadora de propinas

Aquí juntamos todo: entrada, conversión segura, operadores y formato de moneda. Es el primer programa de esta unidad que se siente "de verdad".

```

Console.WriteLine("=== Calculadora de propinas ===");

Console.Write("Total de la cuenta: ");
string entradaCuenta = Console.ReadLine();

Console.Write("Porcentaje de propina (ej. 15): ");
string entradaPorcentaje = Console.ReadLine();

if (decimal.TryParse(entradaCuenta, out decimal cuenta) &&
    decimal.TryParse(entradaPorcentaje, out decimal porcentaje))
{
    decimal propina = cuenta * porcentaje / 100;
    decimal total = cuenta + propina;
    Console.WriteLine($"Propina: {propina:C}");
    Console.WriteLine($"Total a pagar: {total:C}");
}
else
{
    Console.WriteLine("Datos inválidos. Escribe números, por favor.");
}

```

Explicación: pedimos dos datos como texto (ReadLine siempre devuelve string). Luego un solo "if" con dos TryParse unidos por "&&": si AMBOS convierten bien, entramos al bloque que calcula; si alguno falla, vamos al "else" con el mensaje amable. Nota el cortocircuito en acción: si el primer TryParse falla, el segundo ni se evalúa. Dentro del bloque: "cuenta * porcentaje / 100" calcula la propina (con decimal, porque es dinero: exactitud garantizada), "cuenta + propina" da el total, y "{propina:C}" lo muestra como moneda. Prueba romperlo a propósito: escribe "cien" en la cuenta y verás el mensaje amable en lugar de un error técnico. Esa robustez es lo que distingue un programa terminado de un borrador.

(El "if/else" lo veremos formalmente en la próxima unidad; por ahora léelo como "si se pudo convertir, calcula; si no, avisa". Ya lo estás usando antes de estudiarlo a fondo, y eso está bien: así se aprende en contexto.)

Ejemplo 5: var, const y buenas prácticas - ticket de compra

Un ejemplo que modela un ticket de tienda, usando constantes para valores fijos, var donde el tipo es obvio, y nombres descriptivos en todas partes.

```

const decimal IVA = 0.16m;
const string TIENDA = "Abarrotes El Colibrí";

var producto = "Café molido 500g";
var cantidad = 2;
var precioUnitario = 85.50m;

var subtotal = cantidad * precioUnitario;
var impuesto = subtotal * IVA;
var total = subtotal + impuesto;

Console.WriteLine($"*** {TIENDA} ***");
Console.WriteLine($"Producto: {producto}");
Console.WriteLine($"Cantidad: {cantidad} x {precioUnitario:C}");
Console.WriteLine($"Subtotal: {subtotal:C}");
Console.WriteLine($"IVA ({IVA:P0}): {impuesto:C}");
Console.WriteLine($"TOTAL: {total:C}");

```

Explicación: las constantes IVA y TIENDA van en mayúsculas: son valores fijos del negocio (si el IVA cambia, se cambia en UN lugar, no en veinte). "var" se usa porque el tipo es obvio por el valor ("Café..." es string, 2 es int, 85.50m es decimal). "cantidad * precioUnitario": int por decimal da decimal (C# promueve al tipo más preciso). Todo el dinero en decimal, sin excepción. Los nombres ("subtotal", "impuesto", "total") cuentan la historia del cálculo sin necesidad de comentarios. La salida con formato de moneda se ve como un ticket real. Este es el nivel de código limpio al que debes aspirar desde ahora: si alguien lee tu programa sin ejecutarlo, debe entender qué hace.

Ejemplo 6: Incremento, módulo y paridad - turnos de juego

Un ejemplo lúdico que usa ++ y % para algo real: determinar turnos y números pares en un juego.

```

int turno = 1;
Console.WriteLine($"Inicia el turno {turno}");

turno++;
Console.WriteLine($"Avanza al turno {turno}");

turno++;
Console.WriteLine($"Avanza al turno {turno}");

// ¿El turno actual es par? (los pares juegan primero en este juego)
bool turnoPar = turno % 2 == 0;
Console.WriteLine($"¿Turno par? {turnoPar}");

// Contador de puntos con asignación compuesta
int puntos = 0;
puntos += 10;
puntos += 10;
puntos += 5;
Console.WriteLine($"Puntos totales: {puntos}");

```

Explicación: "turno++" avanza el turno de forma compacta (equivalente a "turno = turno + 1"). "turno % 2 == 0": el módulo da el residuo al dividir entre 2; si es 0, el número es par; la comparación produce el bool. "puntos += 10" acumula sin repetir el nombre de la variable dos veces. Son los tres patrones más comunes en juegos y ciclos: avanzar contadores (++), detectar pares/impares o ciclos (%), y acumular totales (+=). Cuando llegues a la unidad de ciclos, verás estos tres en cada ejemplo.

Práctica

8 ejercicios, de menor a mayor dificultad. La regla de oro sigue vigente: intenta primero TÚ (proyecto nuevo, código a mano, ejecutar, verificar), y consulta la solución solo después de intentarlo de verdad. Los ejercicios 7 y 8 son los que más te van a enseñar, porque simulan problemas reales.

1. Ficha personal. Crea un proyecto "Ejercicio1" que declare variables para tu nombre (string), edad (int), estatura en metros (double), tu inicial (char) y si eres estudiante (bool). Imprímelas todas con interpolación en un formato bonito, una por línea.

2. Conversor de monedas. Crea "Ejercicio2": declara una constante con el tipo de cambio (por ejemplo, 17.50 pesos por dólar, en decimal) y una variable con una cantidad en dólares. Calcula e imprime cuántos pesos son, con formato de moneda. Luego cambia la cantidad y ejecuta de nuevo.
3. La trampa del promedio. Crea "Ejercicio3" que calcule el promedio de tres calificaciones enteras (8, 7, 9). Primero hazlo "mal" (división entera) e imprime el resultado; luego hazlo bien (convirtiendo a double o dividiendo entre 3.0) e imprime el resultado correcto. Agrega comentarios explicando la diferencia.
4. Par o impar. Crea "Ejercicio4" que le pida al usuario un número entero (usa TryParse para leerlo de forma segura), determine si es par o impar con el operador módulo, e imprima el resultado. Si la entrada no es válida, muestra un mensaje amable.
5. Calculadora básica. Crea "Ejercicio5" que pida dos números decimales al usuario (TryParse, double) y muestre: suma, resta, multiplicación, división y residuo (módulo). Cuida la división entre cero: si el segundo número es 0, en lugar de dividir muestra "No se puede dividir entre cero".
6. Ticket de compra interactivo. Crea "Ejercicio6" que simule una caja: pide el nombre del producto (string), la cantidad (int con TryParse) y el precio unitario (decimal con TryParse). Calcula subtotal, IVA (16%, como constante) y total. Imprime un ticket con formato de moneda. Si algún dato es inválido, muestra qué dato falló.
7. Depurador. El siguiente programa debería calcular el área de un rectángulo, pero tiene 5 errores (de compilación o de lógica). Cópialo a "Ejercicio7", encuentra y corrige cada uno, con un comentario explicando qué estaba mal:

```
const double base = 10.5;
double altura = "20";
double area = base * altura;
Console.WriteLine("El área es: " + area);
base = 12.0;
```

8. Mi primer programa útil: conversor de temperatura. Crea "Ejercicio8": pide al usuario grados Celsius (double con TryParse), convierte a Fahrenheit con la fórmula $F = C * 9 / 5 + 32$ (¡cuidado con la división entera!) y a Kelvin ($K = C + 273.15$). Muestra los tres valores con 2 decimales. Valida que la temperatura no sea menor al cero absoluto (-273.15 °C): si lo es, muestra un mensaje de error en lugar de calcular.

Soluciones

Solución 1: Ficha personal.

```
string nombre = "Esteban Rivera";
int edad = 28;
double estatura = 1.75;
char inicial = 'E';
bool esEstudiante = true;

Console.WriteLine($"Nombre: {nombre}");
Console.WriteLine($"Edad: {edad} años");
Console.WriteLine($"Estatura: {estatura} m");
Console.WriteLine($"Inicial: {inicial}");
Console.WriteLine($"¿Es estudiante? {esEstudiante}");
```

Explicación: un variable por dato, cada una con el tipo que corresponde a su naturaleza: texto en string, años en int (nadie dice "tengo 28.5 años" en una ficha), estatura en double (requiere decimales), un carácter en char (comillas simples), y el sí/no en bool. La interpolación hace la salida legible. Fíjate que no usé "var" aquí a propósito: en ejercicios de aprendizaje, el tipo explícito refuerza qué tipo es cada cosa.

Solución 2: Conversor de monedas.

```
const decimal TIPO_CAMBIO = 17.50m;
decimal dolares = 100m;

decimal pesos = dolares * TIPO_CAMBIO;
Console.WriteLine($"{dolares:C} dólares son {pesos:C} pesos.");
```

Explicación: el tipo de cambio es "const" porque es un valor fijo de referencia (si cambia mañana, se cambia en un solo lugar) y en mayúsculas por convención. Todo en decimal con sufijo "m" porque es dinero: ni se te ocurra usar double aquí. "{dolares:C}" formatea como moneda. Prueba con 100, con 1 y con 10000: el programa no cambia, solo los datos. Esa es la gracia de las variables: el mismo código sirve para infinitos casos.

Solución 3: La trampa del promedio.

```

int cal1 = 8, cal2 = 9, cal3 = 7;

// FORMA INCORRECTA: división entera trunca el decimal
double promedioMal = (cal1 + cal2 + cal3) / 3;
Console.WriteLine($"Promedio (mal): {promedioMal}"); // 8, no 8.0

// FORMA CORRECTA: al menos un operando decimal antes de dividir
double promedioBien = (cal1 + cal2 + cal3) / 3.0;
Console.WriteLine($"Promedio (bien): {promedioBien}"); // 8.0

// Otra forma correcta: cast explícito antes de la división
double promedioCast = (double)(cal1 + cal2 + cal3) / 3;
Console.WriteLine($"Promedio (cast): {promedioCast}"); // 8.0

```

Explicación: $(8+9+7)=24$, $24/3=8$ exacto... mala suerte con estos números, ¡da igual! Cambia las calificaciones a 8, 7, 9 (suma 24, igual)... prueba con 8, 8, 9: suma 25, $25/3$ entero = 8, pero el real es 8.33. Ahí verás la trampa morder. La lección: en "promedioMal", la división ocurre entre enteros ANTES de guardarse en el double, así que el .33 ya se perdió; en las formas correctas, la conversión ocurre ANTES de dividir (3.0 es double, o el cast a double), así la división es decimal desde el inicio. Regla de oro: convierte antes de operar, no después.

Solución 4: Par o impar.

```

Console.Write("Escribe un número entero: ");
string entrada = Console.ReadLine();

if (int.TryParse(entrada, out int numero))
{
    bool esPar = numero % 2 == 0;
    Console.WriteLine($"El número {numero} es {(esPar ? "par" : "impar")}.");
}
else
{
    Console.WriteLine("Eso no es un número entero válido.");
}

```

Explicación: TryParse valida la entrada sin que el programa se caiga. "numero % 2 == 0": el módulo da el residuo de dividir entre 2; si es 0, es par. El truco "(esPar ? "par" : "impar")" es el operador ternario (un if compacto: condición ? valorSiTrue : valorSiFalse); lo verás a fondo en la unidad de condiciones, aquí lo uso para que la salida sea elegante. Funciona también con negativos: $-4 \% 2 == 0$ da true. Prueba con 0 (es par), con negativos y con texto: el programa nunca se rompe.

Solución 5: Calculadora básica.

```
Console.Write("Primer número: ");
double.TryParse(Console.ReadLine(), out double n1);
Console.Write("Segundo número: ");
double.TryParse(Console.ReadLine(), out double n2);

Console.WriteLine($"Suma: {n1 + n2}");
Console.WriteLine($"Resta: {n1 - n2}");
Console.WriteLine($"Multiplicación: {n1 * n2}");

if (n2 != 0)
{
    Console.WriteLine($"División: {n1 / n2}");
    Console.WriteLine($"Residuo: {n1 % n2}");
}
else
{
    Console.WriteLine("No se puede dividir entre cero.");
}
```

Explicación: leo ambos números con TryParse (si el usuario escribe letras, valen 0 y el programa sigue; en un programa final validaríamos el bool, pero aquí simplifico para enfocarnos en los operadores). Las cuatro operaciones básicas se muestran directo. La división y el módulo van protegidos por "if (n2 != 0)": dividir entre cero con double daría Infinito (no se cae, pero es un resultado sin sentido para una calculadora), así que mejor mostramos el mensaje claro. Esta validación previa ("¿el divisor es cero?") es el tipo de pensamiento defensivo que distingue al profesional: anticipar el caso que rompe tu programa.

Solución 6: Ticket de compra interactivo.

```

const decimal IVA = 0.16m;

Console.Write("Producto: ");
string producto = Console.ReadLine();

Console.Write("Cantidad: ");
bool cantidadOk = int.TryParse(Console.ReadLine(), out int cantidad);

Console.Write("Precio unitario: ");
bool precioOk = decimal.TryParse(Console.ReadLine(), out decimal precio);

if (cantidadOk && precioOk && cantidad > 0 && precio > 0)
{
    decimal subtotal = cantidad * precio;
    decimal impuesto = subtotal * IVA;
    decimal total = subtotal + impuesto;

    Console.WriteLine("----- TICKET -----");
    Console.WriteLine($"Producto: {producto}");
    Console.WriteLine($"Cantidad: {cantidad} x {precio:C}");
    Console.WriteLine($"Subtotal: {subtotal:C}");
    Console.WriteLine($"IVA (16%): {impuesto:C}");
    Console.WriteLine($"TOTAL: {total:C}");
}
else
{
    if (!cantidadOk) Console.WriteLine("Cantidad inválida.");
    if (!precioOk) Console.WriteLine("Precio inválido.");
    if (cantidadOk && cantidad <= 0) Console.WriteLine("La cantidad debe ser mayor a cero.");
    if (precioOk && precio <= 0) Console.WriteLine("El precio debe ser mayor a cero.");
}

```

Explicación: guardo el resultado bool de cada TryParse para saber QUÉ falló y decirlo con precisión (mucho mejor que un genérico "datos inválidos"). La condición del if valida tipo Y rango: no basta que sea número, debe ser positivo (¿cantidad -3? ¿precio 0?). El cálculo usa decimal en todo (dinero) y la constante IVA. El bloque else diagnostica cada problema por separado. Este es el patrón completo de entrada robusta: leer como texto, intentar convertir, validar rangos, calcular solo si todo está bien, e informar errores específicos. Domínalo y podrás escribir cualquier programa interactivo.

Solución 7: Depurador. Código corregido:

```
// Error 1: "base" es palabra reservada de C#; se renombró a baseRectangulo
const double baseRectangulo = 10.5;
// Error 2: no se puede asignar un string a un double; se cambió a número
double altura = 20;
// Error 3 (lógica): faltaba convertir... en realidad el error era el de arriba;
// con altura como double, la multiplicación ya funciona:
double area = baseRectangulo * altura;
Console.WriteLine($"El área es: {area}");
// Error 4: no se puede reasignar una constante; se eliminó la línea
// base = 12.0; <- BORRADA: las constantes no cambian
// Error 5 (estilo): se usó interpolación en lugar de concatenación
```

Explicación de los 5 errores: (1) "base" es palabra reservada del lenguaje (se usa en herencia); el compilador lo rechaza. Moraleja: si un nombre "común" te da error raro, quizá sea reservada; usa otro nombre. (2) "double altura = "20";": no se puede meter texto en una variable numérica directamente; hay que convertir ("double.Parse("20")" o asignar el número 20). (3) En cascada del anterior: con altura como string, "base * altura" no compila (no se multiplica texto). (4) "base = 12.0;" intenta cambiar una const: las constantes se fijan una vez y el compilador lo impide. Si el valor debe cambiar, no es const, es variable normal. (5) La concatenación con "+" funciona, pero la interpolación "\$..." es más legible: mejora de estilo, no error. Si encontraste los 5, tu ojo de depurador ya funciona.

Solución 8: Conversor de temperatura.

```

const double CERO_ABSOLUTO_C = -273.15;

Console.Write("Grados Celsius: ");
if (double.TryParse(Console.ReadLine(), out double celsius))
{
    if (celsius >= CERO_ABSOLUTO_C)
    {
        double fahrenheit = celsius * 9.0 / 5.0 + 32;
        double kelvin = celsius + 273.15;
        Console.WriteLine($"Celsius: {celsius:F2} °C");
        Console.WriteLine($"Fahrenheit: {fahrenheit:F2} °F");
        Console.WriteLine($"Kelvin: {kelvin:F2} K");
    }
    else
    {
        Console.WriteLine("Error: temperatura bajo el cero absoluto (-273.15
°C).");
    }
}
else
{
    Console.WriteLine("Entrada inválida. Escribe un número.");
}
}

```

Explicación: dos niveles de validación anidados: primero ¿es un número? (TryParse), luego ¿está en rango físico posible? ($\geq$ cero absoluto). La fórmula usa 9.0 / 5.0 (decimales) en lugar de 9 / 5 (¡que en enteros daría 1 y arruinaría la conversión!): la trampa de la división entera acecha hasta en la física. "{fahrenheit:F2}" muestra 2 decimales. El cero absoluto es const con nombre descriptivo. Este ejercicio junta TODO lo de la unidad: variables, tipos, operadores, precedencia (la multiplicación antes que la suma en la fórmula), TryParse, interpolación con formato, constantes y validación de rangos. Si lo resolviste sin mirar, la unidad está dominada.

Resumen de la unidad

- Una variable es un espacio de memoria con nombre donde guardas un valor; se declara con tipo, nombre y valor inicial ("int edad = 25;"), y en C# debes inicializarla antes de usarla.
- Los tipos fundamentales: int (enteros), long (enteros grandes), double (decimales aproximados), decimal (decimales exactos, obligatorio para dinero, sufijo m), float (sufijo f), char (un carácter, comillas simples), string (texto, comillas dobles), bool (true/false).

- "var" deja que el compilador deduzca el tipo del valor inicial, pero la variable sigue siendo fuertemente tipada; úsalo cuando el tipo sea obvio, escribe el tipo explícito cuando aclare la intención.
- "const" declara valores fijos que no pueden cambiar (por convención en MAYÚSCULAS); elimina los "números mágicos" y hace el código más legible y seguro.
- Tipos por valor (int, double, decimal, char, bool) se copian al asignar: cada variable es independiente; tipos por referencia (objetos, arreglos) comparten el dato mediante la referencia; string se comporta como valor porque es inmutable.
- Operadores aritméticos: + - * / %; el módulo (%) da el residuo y sirve para paridad y ciclos; los de asignación compuesta (+= -= *= /= %=) acumulan de forma compacta.
- División entera: int / int trunca los decimales (7/2 = 3); para decimales, convierte ANTES de dividir ((double)a / b) o usa operandos decimales (7 / 2.0).
- Operadores de comparación (== != > < >= <=) responden true/false; no confundir "=" (asigna) con "==" (compara); operadores lógicos && (Y), || (O), ! (NO) con cortocircuito.
- ++ y -- incrementan/decrementan en 1; úsalos en su propia línea para evitar ambigüedades; la precedencia se domina con paréntesis generosos, no de memoria.
- El desbordamiento (overflow) de enteros es silencioso por defecto: elige tipos con margen (long) y valida entradas grandes.
- Console.ReadLine() lee texto del usuario y SIEMPRE devuelve string; para números hay que convertir.
- int.Parse convierte pero se cae con texto inválido; Convert sirve para conversiones generales; int.TryParse es la forma profesional para entrada de usuario (devuelve bool, no lanza excepción).
- La interpolación \$"Hola {nombre}" con formatos ({precio:C} moneda, {valor:F2} decimales, {tasa:P0} porcentaje) es la forma moderna y legible de construir mensajes.
- Buenas prácticas: nombres descriptivos en camelCase, decimal para dinero, TryParse para humanos, validar rangos además de tipos, e inicializar al declarar.

UNIDAD 3: Decisiones y ciclos while

En las dos primeras unidades aprendiste lo esencial para sobrevivir en C#: cómo se ve un programa, qué son las variables, qué tipos de datos existen y cómo leer y mostrar información. Hasta aquí, todos los programas que has escrito han sido líneas que se ejecutan en orden, de arriba hacia abajo, sin desviarse ni una sola vez. Es como un camino recto: empiezas en el punto A y terminas en el punto B sin girar.

Pero los programas de verdad no son caminos rectos. Piensa en tu propia vida: no haces lo mismo todos los días sin pensar. Si llueve, tomas un paraguas. Si tienes hambre, cocinas. Si ya es tarde, te vas a dormir. Tomas decisiones todo el tiempo, y además repites acciones: te lavas los dientes todas las mañanas, revisas tu celular varias veces al día, repites tu rutina de la semana. Un programa útil necesita hacer exactamente lo mismo: decidir qué hacer según las circunstancias y repetir tareas sin que tú tengas que escribir cada repetición a mano.

En esta unidad vas a aprender las dos superpotencias que convierten un programa tonto en un programa inteligente: la toma de decisiones y la repetición controlada. Primero dominarás el `if`, el `else if` y el `else`, que son la forma de decirle a la computadora "si pasa esto, haz esto; si no, haz esto otro". Verás cómo combinar condiciones con operadores lógicos para tomar decisiones más ricas, como "si es fin de semana Y no tengo tarea pendiente". También aprenderás el `switch`, que es una forma elegante de manejar muchas opciones al mismo tiempo, como un menú de restaurante donde cada plato tiene su preparación. Y luego entrarás al mundo de los ciclos: el `while` y el `do-while`, que te permiten repetir código mientras se cumpla una condición. Con ellos aprenderás los patrones clásicos que todo programador usa a diario: contadores que llevan la cuenta, acumuladores que suman valores y banderas que marcan si algo ya pasó.

¿Por qué importa esto en la vida real? Porque absolutamente todo el software que usas está lleno de decisiones y ciclos. Cuando inicias sesión en una página, hay un `if` que verifica si tu contraseña es correcta. Cuando ves tu lista de contactos en el celular, hay un ciclo que recorre cada contacto y lo muestra en pantalla. Cuando un videojuego revisa si chocaste contra una pared, es una condición. Cuando una tienda en línea calcula el total de tu carrito, es un acumulador dentro de un ciclo. Sin decisiones y sin ciclos, los programas serían calculadoras primitivas que hacen una sola cosa y se apagan. Con ellos, puedes construir sistemas que reaccionan, que se adaptan, que procesan miles de datos en segundos.

Al terminar esta unidad podrás escribir programas que piensen por sí mismos dentro de lo que tú les enseñaste: un programa que calcule tu calificación final y te diga si aprobaste, un sistema que valide contraseñas, un menú interactivo que no se cierre hasta que el usuario

decida salir, un programa que sume números hasta que tú le digas "basta", y un juego sencillo de adivinar el número donde la computadora te da pistas de "más alto" o "más bajo". Vas a dejar de escribir programas que solo calculan para empezar a escribir programas que deciden y que trabajan por ti.

Esta unidad cubre tres clases: la clase 7 (condicionales if, else if, else con operadores de comparación y lógicos, e ifs anidados), la clase 8 (switch clásico, switch con patrones de coincidencia y switch expressions) y la clase 9 (while, do-while, contadores, acumuladores, banderas y ciclos infinitos). Tómate tu tiempo con cada una. Los errores más comunes de los principiantes en todo el curso viven en esta unidad: confundir el signo = con el ==, olvidar el break en un switch, escribir condiciones que nunca cambian dentro de un while. Los vas a conocer aquí para que nunca te atrapen desprevenido.

Explicación detallada

La computadora que decide: el if

Imagina que sales de tu casa por la mañana. Antes de salir, haces una revisión mental rapidísima: ¿está lloviendo? Si la respuesta es sí, tomas el paraguas. Si la respuesta es no, sales sin él. No lo piensas dos veces, lo haces en automático. El if es exactamente eso, pero para tu programa: una pregunta que se responde con sí o no, y una acción que solo se ejecuta si la respuesta es sí.

La estructura básica es muy simple:

```
if (condicion)
{
    // esto se ejecuta solo si la condicion es verdadera
}
```

La condición es una expresión que se evalúa a un valor booleano: true o false. Ya conoces los booleanos de la unidad anterior. Ahora los vas a usar para controlar el flujo de tu programa. Mira un ejemplo real:

```
int edad = 20;

if (edad >= 18)
{
    Console.WriteLine("Eres mayor de edad, puedes votar.");
}
```

¿Qué pasa aquí? La computadora evalúa `edad >= 18`. Como edad vale 20, la expresión es true, entonces entra al bloque entre llaves y muestra el mensaje. Si edad valiera 15, la expresión sería false y el programa simplemente se saltaría el bloque, como si esas líneas no existieran. El programa continúa después de la llave de cierre sin hacer nada más.

Fíjate en algo importante: las llaves delimitan el bloque de código que depende de la condición. Si solo hay UNA instrucción dentro del if, C# te permite omitir las llaves. Pero aquí va tu primera buena práctica profesional de esta unidad: pon las llaves siempre, aunque haya una sola línea. ¿Por qué? Porque tarde o temprano vas a agregar una segunda línea y si olvidaste las llaves, esa segunda línea se va a ejecutar siempre, haya o no haya condición verdadera. Es uno de esos errores que te hacen perder una hora buscando qué está mal. Los programadores profesionales escriben las llaves siempre. Hazlo tú también desde el primer día.

Operadores de comparación: las preguntas que puedes hacer

Para construir condiciones necesitas los operadores de comparación. Son los signos que te permiten preguntar cómo se relacionan dos valores. Ya los viste de pasada en la unidad 2, pero ahora son protagonistas:

- `==` significa "¿es igual a?" (dos signos de igual, no uno)
- `!=` significa "¿es diferente de?"
- `>` significa "¿es mayor que?"
- `<` significa "¿es menor que?"
- `>=` significa "¿es mayor o igual que?"
- `<=` significa "¿es menor o igual que?"

Cada uno de estos operadores toma dos valores y devuelve un bool: true o false. Así de simple. Mira cómo se usan:


```
int temperatura = 30;

if (temperatura > 25)
{
    Console.WriteLine("Hace calor, prende el ventilador.");
}

string password = "gato123";

if (password == "gato123")
{
    Console.WriteLine("Acceso permitido.");
}
```

Ahora, el error más clásico de todos los principiantes en todos los lenguajes de programación del mundo: confundir `=` con `==`. El signo `=` solo, es asignación: mete un valor dentro de una variable. El doble `==` es comparación: pregunta si dos valores son iguales. Si escribes `if (edad = 18)` en lugar de `if (edad == 18)`, no estás preguntando nada: estás cambiando el valor de edad a 18. En C# moderno el compilador te va a marcar error en muchos de estos casos porque el resultado de una asignación no siempre es un bool válido en un if, pero no te confíes: hay situaciones donde sí compila y el programa se comporta de forma rarísima. La regla de oro: un igual asigna, dos iguales comparan. Repítelo como mantra hasta que salga solo.

Otro detalle con los operadores de comparación: funcionan con números de forma natural, y con strings el `==` compara el contenido del texto. Pero cuidado con comparar números decimales (float, double) con `==`: por la forma en que la computadora guarda los decimales en memoria, dos valores que "deberían" ser iguales a veces difieren en una millonésima. Para decimales es más seguro comparar con rangos, por ejemplo `if (Math.Abs(a - b) < 0.001)`. No te preocupes demasiado por esto ahora, pero guárdalo en la memoria: te va a ahorrar un dolor de cabeza en el futuro.

El camino alternativo: else

Volvamos a la analogía de la mañana: si llueve tomas el paraguas... ¿y si no llueve? Sales sin él, o tomas tus lentes de sol. Siempre hay un plan B. En programación, el plan B se escribe con else:

```
int edad = 15;

if (edad >= 18)
{
    Console.WriteLine("Eres mayor de edad, puedes votar.");
}
else
{
    Console.WriteLine("Todavía no puedes votar, pero ya casi.");
}
```

El else no lleva condición: significa "en cualquier otro caso". Si la condición del if fue true, se ejecuta el bloque del if y se salta el else. Si fue false, se salta el bloque del if y se ejecuta el else. Siempre se ejecuta exactamente uno de los dos bloques, nunca los dos, nunca ninguno. Es una bifurcación perfecta del camino: o vas por la izquierda o vas por la derecha.

Piensa en un semáforo para entender la lógica: si la luz está en verde, avanzas; si no (está en rojo o en amarillo), te detienes. No hay tercera opción. El if-else modela exactamente ese tipo de decisiones binarias de la vida real.

Múltiples caminos: else if

Pero la vida rara vez tiene solo dos opciones. Imagina que vas a calificar un examen: si sacaste 90 o más, tu calificación es A. Si sacaste 80 o más, es B. Si sacaste 70 o más, es C. Si no, reprobaste. Son cuatro caminos posibles, no dos. Para eso existe el else if, que te permite encadenar varias condiciones en orden:

```

int calificacion = 85;
string letra;

if (calificacion >= 90)
{
    letra = "A";
}
else if (calificacion >= 80)
{
    letra = "B";
}
else if (calificacion >= 70)
{
    letra = "C";
}
else
{
    letra = "F (reprobado)";
}

Console.WriteLine("Tu calificación es: " + letra);

```

Aquí hay dos cosas cruciales que entender. Primera: las condiciones se evalúan en orden, de arriba hacia abajo, y en cuanto UNA resulta verdadera, se ejecuta su bloque y todo lo demás se salta. Con calificación = 85, el programa pregunta: ¿85 >= 90? No. ¿85 >= 80? Sí. Asigna "B" y se acabó: ni siquiera revisa las demás condiciones. Segunda: el orden importa muchísimo. Si hubieras puesto primero `if (calificacion >= 70)`, entonces un 95 también cumpliría esa condición y obtendría "C" en lugar de "A". Cuando las condiciones son rangos que se traslapan, siempre ve de lo más específico (o lo más restrictivo) a lo más general. Es como un embudo: primero atrapas los casos especiales y al final dejas caer todo lo demás en el else.

Un truco mental para no equivocarte con el orden: pregúntate "¿esta condición también sería verdadera para casos que quiero atrapar después?". Si la respuesta es sí, tu orden está mal. En el ejemplo de calificaciones, `>= 70` también es verdadero para 95, así que debe ir después de `>= 90` y `>= 80`.

El else final es opcional pero recomendable: actúa como red de seguridad que atrapa todo lo que no entró en ninguna condición anterior. En el ejemplo, atrapa cualquier calificación menor a 70. Piensa en el else final como el "por si acaso" de tu programa. Muchos errores en programas reales ocurren porque el programador asumió que "eso nunca va a pasar" y no puso un else que lo manejara. Ponlo casi siempre.

Operadores lógicos: combinar preguntas

En la vida real casi nunca tomas decisiones con una sola pregunta. No dices "si es sábado, voy a la fiesta". Dices "si es sábado Y ya terminé mi tarea Y tengo dinero, voy a la fiesta". Son tres condiciones que deben cumplirse al mismo tiempo. O a veces es al revés: "si es sábado O es domingo, descanso". Una sola de las dos basta. Los operadores lógicos te permiten combinar condiciones simples en condiciones compuestas:

- `&&` significa Y (AND): el resultado es true solo si AMBAS condiciones son true.
- `||` significa O (OR): el resultado es true si AL MENOS UNA condición es true.
- `!` significa NO (NOT): invierte el valor, convierte true en false y false en true.

Mira cómo se usan en decisiones realistas:

```
int dia = 6; // 6 = sábado, 7 = domingo
bool tareaTerminada = true;
bool tengoDinero = false;

if ((dia == 6 || dia == 7) && tareaTerminada && tengoDinero)
{
    Console.WriteLine("¡A la fiesta!");
}
else
{
    Console.WriteLine("Mejor me quedo en casa.");
}
```

Con `dia = 6`, `tareaTerminada = true` y `tengoDinero = false`: la primera parte `(dia == 6 || dia == 7)` es true (porque `dia` es 6), pero `tengoDinero` es false, y con `&&` basta que una sea false para que todo sea false. Resultado: te quedas en casa. La lógica es implacable y exacta, como debe ser.

Hay dos detalles profesionales que debes conocer sobre `&&` y `||`. El primero se llama cortocircuito (short-circuit): C# evalúa las condiciones de izquierda a derecha y se detiene en cuanto ya sabe el resultado. Con `&&`, si la primera condición es false, ya no evalúa las demás porque el resultado final será false de todos modos. Con `||`, si la primera es true, ya no evalúa las demás. Esto no solo es más rápido: también evita errores. Mira este patrón clásico:

```
string nombre = null;

if (nombre != null && nombre.Length > 3)
{
    Console.WriteLine("Nombre válido.");
}
```

Si `nombre` es `null` y C# evaluara `nombre.Length`, el programa trionaría con un error (`NullReferenceException`). Pero gracias al cortocircuito, como `nombre != null` es `false`, la segunda condición ni se evalúa y el programa sigue tranquilo. Este patrón de "primero verifica que existe, luego úsalo" lo verás en código profesional todo el tiempo. Apréndelo bien.

El segundo detalle es la precedencia: `&&` se evalúa antes que `||`, igual que en matemáticas la multiplicación se hace antes que la suma. Por eso en el ejemplo de la fiesta puse paréntesis alrededor del `||`. Sin paréntesis, la expresión se evaluaría de forma diferente a la que probablemente quieres. La buena práctica es clara: cuando mezcles `&&` y `||`, usa paréntesis siempre para dejar explícita tu intención. Los paréntesis no cuestan nada y evitan que tú (o quien lea tu código dentro de seis meses) tenga que recordar reglas de precedencia.

El operador `!` es el más simple pero también el que más confunde a los principiantes cuando se combina con otros. `!` invierte un booleano: `!true` es `false`, `!false` es `true`. Se usa mucho para expresar "si NO pasa esto":

```
bool lloviendo = false;

if (!lloviendo)
{
    Console.WriteLine("Salgo a caminar.");
}
```

`!lloviendo` se lee como "no está lloviendo". Úsalo con moderación en condiciones complejas: una condición con dos o tres negaciones (`if (!(a && !b) || !c)`) es una pesadilla de leer. Si tu condición necesita tantas negaciones, probablemente conviene reescribirla de forma positiva o partirla en variables booleanas con nombres claros:

```
bool esFinDeSemana = (dia == 6 || dia == 7);
bool puedoSalir = esFinDeSemana && tareaTerminada && tengoDinero;

if (puedoSalir)
{
    Console.WriteLine("¡A la fiesta!");
}
```

¿Ves la diferencia? La condición final se lee como español: "si puedo salir". Las variables booleanas intermedias con buenos nombres son una de las mejores herramientas para escribir condiciones legibles. Úsalas sin miedo: no hacen tu programa más lento y lo hacen muchísimo más claro.

Ifs anidados: decisiones dentro de decisiones

A veces una decisión depende del resultado de otra. Imagina que vas a un restaurante: primero decides si quieres comer ahí (¿hay mesa libre?). Si sí, entonces decides qué pedir (¿quiero pizza o pasta?). La segunda decisión solo existe si la primera fue afirmativa. Eso es un if anidado: un if dentro de otro if.

```
bool hayMesaLibre = true;
bool quieroPizza = false;

if (hayMesaLibre)
{
    Console.WriteLine("Nos sentamos.");

    if (quieroPizza)
    {
        Console.WriteLine("Pido una pizza.");
    }
    else
    {
        Console.WriteLine("Pido una pasta.");
    }
}
else
{
    Console.WriteLine("Buscamos otro restaurante.");
}
```

Los ifs anidados son perfectamente válidos y a veces necesarios, pero aquí viene una advertencia seria de programador experimentado: la anidación profunda es el enemigo de la legibilidad. Cada nivel de anidación es como una muñeca rusa: tienes que abrir una para llegar a la siguiente, y después de tres o cuatro niveles ya nadie entiende qué está pasando. Mira este horror:

```
if (usuarioValido)
{
    if (passwordCorrecto)
    {
        if (cuentaActiva)
        {
            if (tienePermiso)
            {
                Console.WriteLine("Bienvenido al sistema.");
            }
        }
    }
}
```

Esto se conoce como "código espagueti" o, en broma entre programadores, "el infierno de las flechas" por la forma que toma el código. ¿Cómo se evita? Con una técnica llamada early return (retorno temprano) o, más generalmente, "salir temprano de los casos no válidos". En lugar de anidar todo lo bueno dentro de condiciones positivas, primero descartas lo malo:

```
if (!usuarioValido)
{
    Console.WriteLine("Usuario no válido.");
    return;
}

if (!passwordCorrecto)
{
    Console.WriteLine("Contraseña incorrecta.");
    return;
}

// si llegamos aquí, todo está bien
Console.WriteLine("Bienvenido al sistema.");
```

¿Ves qué diferente se siente? Cada validación es independiente, el código fluye de arriba hacia abajo sin sangrías monstruosas, y es obvio qué pasa en cada caso. (El `return` termina

el método actual; lo verás a fondo en la Unidad 4 cuando hablemos de métodos. Por ahora, piensa en él como "aquí se acaba, no sigas".)

La regla práctica: si tienes más de dos niveles de anidación, detente y pregúntate si puedes reestructurar con early returns o con else if. Tu yo del futuro te lo va a agradecer.

El operador ternario: el if de una línea

Hay un atajo para el caso más simple de todos: cuando quieres asignar un valor u otro según una condición. Se llama operador ternario (o condicional) y se escribe con `?` y `:`:

```
int edad = 20;
string mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
Console.WriteLine(mensaje);
```

Se lee así: "si edad >= 18 es verdadero, usa 'Mayor de edad'; si no, usa 'Menor de edad'". Es compacto y elegante para asignaciones simples. Pero tiene una regla de uso estricta: úsalo SOLO cuando la expresión sea corta y obvia. En cuanto el ternario crece, se vuelve ilegible:

```
// NO hagas esto:
string resultado = (a > b) ? ((a > c) ? "a es mayor" : "c es mayor") : ((b > c) ?
    "b es mayor" : "c es mayor");
```

Si tu ternario necesita ternarios anidados, escríbelo como if-else normal. La brevedad no vale nada si nadie (incluyéndote a ti dentro de un mes) puede entender qué hace el código.

El switch: cuando hay muchas opciones

El if-else if-else funciona bien para unas cuantas condiciones, pero cuando tienes muchas opciones basadas en el MISMO valor, se vuelve repetitivo y tedioso. Imagina un menú de restaurante con 10 platillos: escribir 10 else if comparando la misma variable es cansado de escribir y cansado de leer. Para esos casos existe el switch, que es como decirle a la computadora: "mira este valor, y según lo que sea, ve directo a la opción que le corresponde".


```

int diaSemana = 3;

switch (diaSemana)
{
    case 1:
        Console.WriteLine("Lunes: toca pierna en el gym.");
        break;
    case 2:
        Console.WriteLine("Martes: toca pecho y tríceps.");
        break;
    case 3:
        Console.WriteLine("Miércoles: toca espalda y bíceps.");
        break;
    case 4:
        Console.WriteLine("Jueves: toca hombro.");
        break;
    case 5:
        Console.WriteLine("Viernes: toca pierna otra vez.");
        break;
    default:
        Console.WriteLine("Fin de semana: descanso activo.");
        break;
}

```

La estructura: `switch` recibe una expresión (aquí, la variable `diaSemana`). Cada `case` es un valor posible. Si el valor coincide con un case, se ejecuta el código de ese case. El `default` es el equivalente al else final: atrapa todo lo que no coincidió con ningún case.

Ahora, la palabra mágica que NO debes olvidar: `break`. Cada case debe terminar con break (o con otra instrucción de salto como return o throw). ¿Por qué? Porque sin break, C#... bueno, en realidad C# es estricto aquí: a diferencia de otros lenguajes como C o Java, en C# el compilador NO te permite que la ejecución "caiga" de un case al siguiente por accidente. Si olvidas el break, el programa ni siquiera compila y te marca error. Esto es una protección deliberada del lenguaje contra uno de los errores más comunes de la historia de la programación. Aun así, debes entender el concepto: el break le dice al switch "ya terminé, sal del switch". Sin él, en otros lenguajes la ejecución continuaría con el siguiente case aunque no coincidiera, causando comportamientos extrañísimos.

Puedes agrupar varios cases que comparten el mismo código simplemente apilándolos:

```

switch (diaSemana)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Día laboral: a trabajar.");
        break;
    case 6:
    case 7:
        Console.WriteLine("Fin de semana: a descansar.");
        break;
    default:
        Console.WriteLine("Día inválido.");
        break;
}

```

Esto es elegante: cinco cases caen al mismo código sin repetirlo. Es una de las pocas formas de "caída" que C# permite, y es intencional y clara.

El switch funciona con varios tipos: int, string, char, enum y otros. Con strings es muy útil para menús de texto:

```

Console.Write("Elige una opción (pizza/tacos/ensalada): ");
string eleccion = Console.ReadLine();

switch (eleccion)
{
    case "pizza":
        Console.WriteLine("Marchando una pizza grande.");
        break;
    case "tacos":
        Console.WriteLine("Órden de tacos al pastor en camino.");
        break;
    case "ensalada":
        Console.WriteLine("Ensalada fresca, buena elección.");
        break;
    default:
        Console.WriteLine("Esa opción no está en el menú.");
        break;
}

```

Un detalle importante con strings: la comparación es sensible a mayúsculas y minúsculas. Si el usuario escribe "Pizza" con mayúscula, no va a coincidir con el case "pizza". Para evitarlo, puedes normalizar la entrada con ``eleccion.ToLower()`` antes del switch, o usar patrones más avanzados que veremos ahora.

¿Cuándo usar switch y cuándo usar if-else? La regla general: usa switch cuando estás comparando UNA sola variable o expresión contra una lista de valores constantes. Usa if-else cuando cada condición es diferente (rangos, combinaciones lógicas, condiciones sobre variables distintas). Si te encuentras escribiendo ``if (x == 1) ... else if (x == 2) ... else if (x == 3)...``, es señal de que un switch quedaría mejor.

Switch moderno: patrones de coincidencia

Hasta aquí el switch que viste es el clásico, que compara valores exactos. Pero C# moderno (desde la versión 7 en adelante, y tú estás usando .NET 8 o superior) le dio superpoderes al switch con algo llamado pattern matching, o patrones de coincidencia. La idea es que los cases ya no solo comparan igualdad: pueden verificar tipos, rangos y condiciones personalizadas. Es como si el portero de un antro ya no solo revisara tu nombre en la lista, sino también tu edad, tu vestimenta y si vienes en la lista de invitados especiales, todo al mismo tiempo.

El primer superpoder es la cláusula ``when``, que agrega una condición extra a un case:

```
int edad = 20;
string categoria;

switch (edad)
{
    case int e when e < 13:
        categoria = "Niño";
        break;
    case int e when e < 18:
        categoria = "Adolescente";
        break;
    case int e when e < 65:
        categoria = "Adulto";
        break;
    default:
        categoria = "Adulto mayor";
        break;
}
```

¿Qué está pasando aquí? ``case int e when e < 13`` significa: "si el valor es un int (llámalo e de cariño) Y además e es menor que 13". La variable ``e`` captura el valor para que puedas usarlo en la condición del `when`. Esto combina lo mejor de los dos mundos: la estructura limpia del `switch` con la flexibilidad de las condiciones del `if`. Fíjate que el orden sigue importando igual que en el `else if`: los rangos van de menor a mayor para que cada valor caiga en la primera categoría que le corresponde.

El segundo superpoder son los patrones de tipo (type patterns): el `switch` puede recibir un objeto de cualquier tipo y cada `case` verifica de qué tipo es:

```
object dato = "Hola, mundo";

switch (dato)
{
    case int numero:
        Console.WriteLine("Es un entero: " + numero);
        break;
    case string texto:
        Console.WriteLine("Es un texto de " + texto.Length + " letras.");
        break;
    case double decimal_:
        Console.WriteLine("Es un decimal: " + decimal_);
        break;
    case null:
        Console.WriteLine("No hay nada aquí.");
        break;
    default:
        Console.WriteLine("Es algo raro que no esperaba.");
        break;
}
```

Esto es increíblemente útil en la vida real. Imagina que tu programa recibe datos de diferentes fuentes: a veces un número, a veces un texto, a veces nada. Con type patterns manejas cada caso de forma limpia, y además la variable capturada (``numero``, ``texto``) ya viene con el tipo correcto, lista para usar sin conversiones manuales. El ``case null`` merece mención especial: verifica explícitamente si el valor es nulo, algo que en código profesional se hace constantemente para evitar errores.

También puedes combinar patrones con propiedades usando patrones de propiedades, aunque eso lo verás más a fondo cuando estudies clases y objetos. Por ahora quédate con la idea: el `switch` moderno es mucho más que una lista de igualdades.

Switch expressions: el switch que devuelve un valor

Hay una forma aún más compacta y moderna de escribir un switch cuando lo único que quieres es calcular un valor según una variable. Se llama switch expression y se ve así:

```
int diaSemana = 3;

string actividad = diaSemana switch
{
    1 => "Pierna en el gym",
    2 => "Pecho y tríceps",
    3 => "Espalda y bíceps",
    4 => "Hombro",
    5 => "Pierna otra vez",
    6 or 7 => "Descanso",
    _ => "Día inválido"
};

Console.WriteLine("Hoy toca: " + actividad);
```

Analicemos la sintaxis porque es nueva: después de la variable pones la palabra `switch`, y dentro de las llaves cada opción usa `=>` (flecha) en lugar de `case` y dos puntos. No hay breaks porque cada rama produce directamente un valor. El símbolo `_` (guion bajo) es el comodín que reemplaza al default: significa "cualquier otro valor". Y `6 or 7` muestra cómo combinar múltiples patrones con la palabra `or` (también existe `and` y `not`).

Las switch expressions tienen dos ventajas grandes. Primera: son expresiones, no instrucciones. Eso significa que producen un valor que puedes asignar directamente a una variable, como en el ejemplo. No necesitas declarar la variable antes y asignarla dentro de cada case. Segunda: el compilador te ayuda más. Si el switch expression no cubre todos los casos posibles y no tiene comodín `_`, el compilador te avisa. Con el switch clásico, olvidar el default solo significa que a veces no pasa nada; con la switch expression, el compilador te obliga a ser exhaustivo. Esa ayuda del compilador vale oro: convierte errores que descubrirías hasta que el programa falla en errores que ves mientras escribes.

También puedes usar `when` en las switch expressions:

```
string categoria = edad switch
{
    < 13 => "Niño",
    < 18 => "Adolescente",
    < 65 => "Adulto",
    _ => "Adulto mayor"
};
```

¿Viste qué limpio quedó? Los patrones relacionales (`< 13`, `< 18`) comparan directamente sin necesidad de declarar variable. Esta es la forma moderna y preferida de escribir este tipo de lógica en C# actual. Cuando veas código profesional moderno, te vas a encontrar switch expressions por todos lados.

Un consejo práctico: no te sientas obligado a usar la sintaxis más moderna en todo. El switch clásico con breaks sigue siendo válido y a veces más claro cuando cada case hace varias acciones (no solo calcular un valor). La switch expression brilla cuando cada rama produce un valor. Elige la herramienta según el trabajo: ¿múltiples acciones por caso? switch clásico. ¿un valor por caso? switch expression.

Ciclos: el poder de repetir

Ahora cambiamos de tercio. Ya sabes tomar decisiones; es hora de aprender a repetir. Un ciclo (o bucle, o loop) es una estructura que ejecuta un bloque de código una y otra vez mientras se cumpla una condición. Es la diferencia entre escribir:

```
Console.WriteLine("1");
Console.WriteLine("2");
Console.WriteLine("3");
// ... así hasta el 100, qué flojera
```

Y escribir:

```
int numero = 1;
while (numero <= 100)
{
    Console.WriteLine(numero);
    numero = numero + 1;
}
```

Cuatro líneas hacen el trabajo de cien. Y si mañana quieres contar hasta un millón, cambias un solo número. Esa es la magia de los ciclos: convierten trabajo repetitivo en trabajo

inteligente. Piensa en una receta de cocina: "bate la mezcla durante 5 minutos" no te dice "mueve la cuchara, mueve la cuchara, mueve la cuchara..." quinientas veces. Te da la acción y la condición de paro. Los ciclos son las recetas de la programación.

El while: repetir mientras se cumpla la condición

El while es el ciclo más fundamental. Su lógica es: "mientras la condición sea verdadera, repite este bloque". Antes de cada repetición (incluyendo la primera), evalúa la condición. Si es true, ejecuta el bloque y vuelve a preguntar. Si es false, sale del ciclo y continúa con el resto del programa.

```
int contador = 1;

while (contador <= 5)
{
    Console.WriteLine("Vuelta número " + contador);
    contador = contador + 1;
}

Console.WriteLine("Ciclo terminado.");
```

Vamos a seguir la ejecución paso a paso, porque entender este flujo es crucial:

1. contador empieza en 1.
2. ¿1 <= 5? Sí. Muestra "Vuelta número 1". contador ahora vale 2.
3. ¿2 <= 5? Sí. Muestra "Vuelta número 2". contador ahora vale 3.
4. ¿3 <= 5? Sí. Muestra "Vuelta número 3". contador ahora vale 4.
5. ¿4 <= 5? Sí. Muestra "Vuelta número 4". contador ahora vale 5.
6. ¿5 <= 5? Sí. Muestra "Vuelta número 5". contador ahora vale 6.
7. ¿6 <= 5? No. Sale del ciclo.
8. Muestra "Ciclo terminado."

Fíjate en algo esencial: la condición se revisa ANTES de cada vuelta, incluyendo la primera. Si la condición es falsa desde el inicio, el bloque no se ejecuta ni una sola vez:

```
int contador = 10;

while (contador <= 5) // 10 <= 5 es false desde el inicio
{
    Console.WriteLine("Esto nunca se muestra.");
}
```

El programa simplemente se salta el ciclo completo. Esto es importante cuando la condición depende de datos que vienen del usuario o de un archivo: tu ciclo debe comportarse bien incluso si no hay nada que procesar.

La línea `contador = contador + 1` merece atención especial. No es una ecuación matemática (en matemáticas sería imposible: ningún número es igual a sí mismo más uno). Es una instrucción: "toma el valor actual de contador, súmalo 1, y guarda el resultado de nuevo en contador". Es tan común que C# tiene atajos: `contador++` significa exactamente `contador = contador + 1` (incremento en 1), y `contador--` significa `contador = contador - 1` (decremento en 1). También existen `+=`, `-=`, `*=` y `/=` para otros casos: `total += 5` es lo mismo que `total = total + 5`. Úsalos: son estándar en código profesional.

El peligro real: ciclos infinitos

Ahora hablemos del error más temido (y más común) al usar `while`: el ciclo infinito. Ocurre cuando la condición del ciclo nunca se vuelve falsa, entonces el programa repite el bloque para siempre. Mira el ejemplo clásico:

```
int contador = 1;

while (contador <= 5)
{
    Console.WriteLine("Vuelta número " + contador);
    // ¡OLVIDAMOS incrementar el contador!
}
```

contador siempre vale 1, 1 siempre es `<= 5`, así que el ciclo nunca termina. El programa se queda imprimiendo "Vuelta número 1" eternamente hasta que lo detengas a la fuerza (Ctrl+C en la consola). En un programa real, un ciclo infinito puede congelar una aplicación o un servidor.

¿Cómo evitarlos? Con una regla de oro: dentro del cuerpo del ciclo, SIEMPRE debe haber algo que acerque la condición a volverse falsa. Normalmente es modificar la variable que aparece en la condición: incrementar un contador, leer un nuevo valor del usuario, avanzar a

un siguiente elemento. Antes de escribir un while, pregúntate: "¿qué va a hacer que esto termine?". Si no tienes una respuesta clara, no escribas el ciclo todavía.

Otra causa común de ciclos infinitos es una condición mal planteada:

```
int numero = 10;

while (numero != 0)
{
    numero = numero - 2; // 10, 8, 6, 4, 2, 0... aquí sí termina
}
```

Este sí termina. Pero si empezaras en 11: 11, 9, 7, 5, 3, 1, -1, -3... nunca toca exactamente el 0, ¡ciclo infinito! Cuando uses `!=` en la condición de un ciclo, asegúrate de que el valor realmente vaya a alcanzar ese número exacto. En caso de duda, usa `<`, `<=`, `>` o `>=`, que son más seguros porque no dependen de tocar un valor exacto.

El do-while: primero actúa, luego pregunta

El while pregunta antes de actuar. Su primo, el do-while, actúa primero y pregunta después. La diferencia parece pequeña pero es importante: con do-while, el bloque se ejecuta AL MENOS UNA VEZ, garantizado, porque la condición se evalúa al final.

```
int numero;

do
{
    Console.WriteLine("Adivina el número (1-10): ");
    numero = int.Parse(Console.ReadLine());

    if (numero < 7)
    {
        Console.WriteLine("Más alto...");
    }
    else if (numero > 7)
    {
        Console.WriteLine("Más bajo...");
    }
} while (numero != 7);

Console.WriteLine("¡Correcto! El número era 7.");
```

Piensa en cuándo tiene sentido: aquí necesitas pedir el número al menos una vez antes de poder comparar si es correcto. Con un while normal tendrías que pedir el número una vez antes del ciclo y luego otra vez dentro del ciclo, repitiendo código. El do-while evita esa duplicación: "haz esto, y repite mientras no sea correcto".

La sintaxis tiene un detalle que no debes olvidar: el do-while termina con punto y coma después de la condición: ``}` while (condicion);`. Es el único ciclo que lleva punto y coma al final, y si lo olvidas el compilador se queja. Es fácil de recordar si piensas que el do-while es como una instrucción larga que empieza en `do` y termina en el `;`.`

¿Cuándo usar while y cuándo do-while? La regla es simple: si el bloque debe ejecutarse al menos una vez sí o sí (mostrar un menú, pedir un dato, lanzar los dados la primera vez), usa do-while. Si es posible que no deba ejecutarse ninguna vez (procesar elementos de una lista que podría estar vacía, reintentar solo si algo falló antes), usa while. En la práctica verás mucho más el while que el do-while, pero cuando necesites el do-while, nada más lo sustituye con elegancia.

Los tres patrones clásicos: contador, acumulador y bandera

Con los ciclos vienen tres patrones (formas típicas de resolver problemas) que vas a usar en prácticamente todos tus programas. Apréndetelos de memoria porque son el abecedario de la lógica de ciclos.

El CONTADOR lleva la cuenta de cuántas veces pasó algo. Empieza en cero (o en uno, según convenga) y se incrementa en 1 cada vez:

```

int aprobados = 0;
int i = 1;

while (i <= 10)
{
    Console.WriteLine("Calificación del alumno " + i + ": ");
    int calif = int.Parse(Console.ReadLine());

    if (calif >= 70)
    {
        aprobados = aprobados + 1; // o aprobados++
    }

    i++;
}

Console.WriteLine("Aprobaron " + aprobados + " de 10 alumnos.");

```

El contador responde a la pregunta "¿cuántos?". ¿Cuántos aprobaron? ¿Cuántos intentos hizo el usuario? ¿Cuántos números pares hay? Siempre que necesites contar ocurrencias, es un contador: inicialízalo en 0, súmale 1 cuando ocurra lo que cuentas.

El ACUMULADOR suma (o concatena, o acumula de otra forma) valores a lo largo del ciclo. Empieza en cero (o en un valor inicial) y en cada vuelta le agrega algo:

```

int total = 0;
int cantidad = 1;

while (cantidad <= 5)
{
    Console.WriteLine("Precio del producto " + cantidad + ": ");
    int precio = int.Parse(Console.ReadLine());
    total = total + precio; // o total += precio
    cantidad++;
}

Console.WriteLine("Total de la compra: $" + total);

```

El acumulador responde a la pregunta "¿cuánto en total?". El total de una compra, la suma de calificaciones para luego sacar promedio, el texto completo formado por pedazos. La diferencia clave con el contador: el contador suma siempre 1 (cuenta eventos), el acumulador suma valores variables (acumula cantidades). Un error típico es inicializar mal el acumulador: para sumas empieza en 0, pero para multiplicaciones (acumulador multiplicativo) debe

empezar en 1, porque todo número multiplicado por 0 da 0. Si estás calculando un factorial y empiezas en 0, tu resultado siempre será 0 y te vas a frustrar un buen rato hasta descubrirlo.

La BANDERA (flag) es una variable booleana que marca si algo ya ocurrió. Empieza en false y cambia a true cuando sucede el evento que estás vigilando:

```
bool encontrado = false;
int intento = 1;

while (intento <= 3 && !encontrado)
{
    Console.WriteLine("Intento " + intento + ": escribe la palabra secreta: ");
    string palabra = Console.ReadLine();

    if (palabra == "colibri")
    {
        encontrado = true;
    }

    intento++;
}

if (encontrado)
{
    Console.WriteLine("¡Acceso concedido!");
}
else
{
    Console.WriteLine("Demasiados intentos. Bloqueado.");
}
```

La bandera responde a la pregunta "¿pasó o no pasó?". ¿Se encontró el dato? ¿El usuario ya inició sesión? ¿Hubo algún error? Fíjate cómo la bandera controla el ciclo: `while (intento <= 3 && !encontrado)` significa "sigue intentando mientras queden intentos Y todavía no se haya encontrado". En cuanto la bandera se vuelve true, el ciclo termina aunque queden intentos. Es un patrón elegantísimo para búsquedas: deja de buscar en cuanto encuentras lo que buscabas, sin vueltas desperdiciadas.

Estos tres patrones se combinan todo el tiempo en programas reales. Un programa de calificaciones usa un contador para saber cuántos alumnos van, un acumulador para sumar sus calificaciones, y al final divide para obtener el promedio. Un juego usa banderas para saber si el jugador ganó o perdió. Domina estos tres patrones y podrás resolver el 80% de los problemas con ciclos que te encuentres.

Buenas prácticas con decisiones y ciclos

Antes de pasar a los ejemplos, consolidemos las buenas prácticas que separan al código de principiante del código profesional:

1. Condiciones legibles: escribe condiciones que se lean como frases. ``if (edad >= 18)`` se lee perfecto. ``if (!(e < 18))`` significa lo mismo pero cuesta el doble de esfuerzo mental. Si una condición es compleja, guárdala en una variable booleana con un buen nombre antes del `if`.
2. Early return / salida temprana: valida primero los casos inválidos y sal, en lugar de anidar todo el código bueno dentro de condiciones. Reduce la anidación y hace el flujo obvio.
3. Evita la anidación profunda: más de dos niveles de `ifs` anidados es señal de que debes reestructurar. Extrae lógica a variables booleanas, usa early returns o reorganiza las condiciones.
4. El `else` final como red de seguridad: casi siempre pon un `else` o un `default` al final. Atrapa los casos inesperados y te avisa cuando algo raro pasa, en lugar de fallar en silencio.
5. En ciclos, garantiza la salida: cada `while` debe tener algo dentro que acerque la condición a terminarse. Pregúntate siempre "¿qué hace que esto termine?" antes de escribir el ciclo.
6. Nombres claros para contadores y banderas: ``intentos``, ``totalCompra``, ``encontrado``, ``sigueJugando`` dicen mucho más que ``x``, ``t`` o ``flag``. El nombre debe revelar el propósito.
7. Inicializa tus patrones: contador en 0, acumulador de suma en 0, acumulador de multiplicación en 1, bandera en `false`. La inicialización incorrecta es fuente de errores silenciosos (el programa corre pero da resultados mal).

Ejemplos paso a paso

Ejemplo 1: Clasificador de calificaciones con `if-else if-else`

Problema: pedir la calificación numérica de un alumno (0 a 100) y mostrar su letra (A, B, C, D, F) más un mensaje. Además, validar que el número esté en rango.

```

Console.Write("Escribe tu calificación (0-100): ");
int calif = int.Parse(Console.ReadLine());

string letra;
string mensaje;

if (calif < 0 || calif > 100)
{
    letra = "Inválida";
    mensaje = "La calificación debe estar entre 0 y 100.";
}
else if (calif >= 90)
{
    letra = "A";
    mensaje = "¡Excelente! Sigue así.";
}
else if (calif >= 80)
{
    letra = "B";
    mensaje = "Muy bien, casi perfecto.";
}
else if (calif >= 70)
{
    letra = "C";
    mensaje = "Aprobado, pero puedes mejorar.";
}
else if (calif >= 60)
{
    letra = "D";
    mensaje = "Apenas pasaste, échale más ganas.";
}
else
{
    letra = "F";
    mensaje = "Reprobado. A estudiar más.";
}

Console.WriteLine("Calificación: " + letra);
Console.WriteLine(mensaje);

```

Explicación línea por línea:

- `Console.Write("Escribe tu calificación (0-100): ");` muestra el mensaje pidiendo el dato.
Uso Write (sin salto de línea) para que el usuario escriba en el mismo renglón.

- `int calif = int.Parse(Console.ReadLine());` lee lo que escribe el usuario y lo convierte a entero. Recuerda: `ReadLine` siempre devuelve string, por eso necesitamos `Parse`.
- `string letra;` y `string mensaje;` declaran dos variables de texto sin valor inicial. Las vamos a llenar dentro de las ramas del `if`.
- `if (calif < 0 || calif > 100)` es la primera validación: si el número está fuera del rango válido, lo marcamos como inválido. Fíjate que la validación va PRIMERO, antes de clasificar. Esta es la técnica de "valida la entrada antes de procesarla": si los datos están mal, no tiene sentido seguir.
- El operador `||` significa que basta que UNA de las dos condiciones sea verdadera: menor que 0 O mayor que 100.
- `else if (calif >= 90)` y los siguientes clasifican en orden descendente: primero el rango más alto. Como vimos, el orden es crítico: si `>= 60` estuviera antes que `>= 90`, un 95 caería en "D".
- El `else` final atrapa todo lo menor a 60: la F. No necesita condición porque es "todo lo demás".
- Al final mostramos la letra y el mensaje. Como todas las ramas asignan valores a `letra` y `mensaje`, el compilador está contento: las variables siempre tendrán valor al llegar al `WriteLine`.

Prueba mental: si el usuario escribe 85, la primera condición (fuera de rango) es false, `>= 90` es false, `>= 80` es true: letra = "B", mensaje = "Muy bien...". Las demás ramas se saltan. Si escribe 150: la primera es true, marca inválida y se salta todo lo demás.

Ejemplo 2: Menú interactivo con switch y do-while

Problema: mostrar un menú de opciones que se repita hasta que el usuario elija salir. Este ejemplo combina `do-while` (el menú debe mostrarse al menos una vez) con `switch` (varias opciones sobre la misma variable).

```

int opcion;

do
{
    Console.WriteLine("===== MENÚ PRINCIPAL =====");
    Console.WriteLine("1. Saludar");
    Console.WriteLine("2. Mostrar la fecha de hoy");
    Console.WriteLine("3. Contar hasta 5");
    Console.WriteLine("0. Salir");
    Console.Write("Elige una opción: ");

    opcion = int.Parse(Console.ReadLine());
    Console.WriteLine(); // línea en blanco para separar

    switch (opcion)
    {
        case 1:
            Console.WriteLine("¡Hola! Qué bueno verte por aquí.");
            break;
        case 2:
            Console.WriteLine("Hoy es: " + DateTime.Now.ToShortDateString());
            break;
        case 3:
            int n = 1;
            while (n <= 5)
            {
                Console.Write(n + " ");
                n++;
            }
            Console.WriteLine();
            break;
        case 0:
            Console.WriteLine("Nos vemos pronto. ¡Bye!");
            break;
        default:
            Console.WriteLine("Opción no válida, intenta de nuevo.");
            break;
    }

    Console.WriteLine(); // otra línea en blanco

} while (opcion != 0);

Console.WriteLine("Programa terminado.");

```


Explicación línea por línea:

- `int opcion;` declara la variable que guardará la elección del usuario. La declaramos ANTES del do-while porque la condición del while (al final) necesita verla. Si la declaráramos dentro del bloque, no existiría fuera de él (ámbito de variables, que verás a fondo en la Unidad 4).
- `do {` inicia el bloque que se ejecutará al menos una vez.
- Las líneas de `Console.WriteLine` dibujan el menú en pantalla. Los `=====` son decoración simple para que se vea como un menú de verdad.
- `opcion = int.Parse(Console.ReadLine());` lee la elección del usuario.
- `Console.WriteLine();` sin argumentos imprime una línea vacía: un truco simple para separar visualmente las secciones.
- El `switch (opcion)` dirige a cada opción: case 1 saluda, case 2 muestra la fecha usando `DateTime.Now` (la fecha y hora actuales del sistema), case 3 cuenta hasta 5, case 0 se despide.
- En el case 3 hay un while ANIDADO dentro del switch: un ciclo dentro de una opción del menú. `while (n <= 5)` imprime los números del 1 al 5 en la misma línea (usando `Write` en lugar de `WriteLine`) y luego un `WriteLine` vacío para el salto de línea. Esto demuestra que puedes combinar estructuras: un ciclo dentro de un switch dentro de un ciclo.
- El `default` atrapa cualquier número que no sea 0, 1, 2 o 3, y le dice al usuario que intente de nuevo. Sin este default, una opción inválida simplemente no haría nada y el menú se repetiría sin explicación: confuso para el usuario.
- `} while (opcion != 0);` es la condición de repetición: mientras la opción NO sea 0, vuelve a mostrar el menú. Cuando el usuario elige 0, el ciclo termina. No olvides el punto y coma final: es obligatorio en do-while.
- El último `WriteLine` confirma que el programa terminó limpiamente.

Este patrón de "menú con do-while + switch" es uno de los más usados en programas de consola reales: sistemas de inventario, cajeros automáticos simulados, juegos de texto. Si lo dominas, ya sabes estructurar una gran familia de programas.

Ejemplo 3: Contador y acumulador - promedio de calificaciones

Problema: pedir calificaciones una por una hasta que el usuario escriba -1 para terminar, y al final mostrar cuántas capturó y el promedio. Aquí se combinan los tres patrones: contador (cuántas), acumulador (suma total) y la condición de salida (el -1).

```

int suma = 0;          // acumulador: va sumando las calificaciones
int cantidad = 0;     // contador: cuenta cuántas calificaciones van

Console.WriteLine("Escribe calificaciones (0-100). Escribe -1 para terminar.");

while (true)
{
    Console.Write("Calificación: ");
    int calif = int.Parse(Console.ReadLine());

    if (calif == -1)
    {
        break; // el usuario quiere terminar: salimos del ciclo
    }

    if (calif < 0 || calif > 100)
    {
        Console.WriteLine("Calificación inválida, no se cuenta. Intenta de nuevo.");
        continue; // salta a la siguiente vuelta sin contar esta
    }

    suma = suma + calif; // acumulamos
    cantidad = cantidad + 1; // contamos una más
}

if (cantidad > 0)
{
    double promedio = (double)suma / cantidad;
    Console.WriteLine("Capturaste " + cantidad + " calificaciones.");
    Console.WriteLine("Promedio: " + promedio);
}
else
{
    Console.WriteLine("No capturaste ninguna calificación.");
}

```

Explicación línea por línea:

- `int suma = 0;` inicializa el acumulador en 0. Regla de oro: acumulador de suma siempre empieza en 0.
- `int cantidad = 0;` inicializa el contador en 0. Contador siempre empieza en 0.

- ``while (true)`` es un ciclo que, en principio, nunca terminaría solo: la condición siempre es verdadera. Esto es intencional y se llama ciclo con salida interna: en lugar de poner la condición de salida en el while, la manejamos con break adentro. Es un patrón válido cuando la condición de salida se decide EN MEDIO del bloque (después de leer el dato), no al inicio ni al final.
- Dentro del ciclo, leemos la calificación.
- ``if (calif == -1) { break; }``: si el usuario escribió -1, el ``break`` rompe el ciclo inmediatamente y salta a la primera línea después del while. El -1 se conoce como valor centinela: un valor especial que no es un dato real, sino una señal de "ya terminé". Es un patrón clásico: eliges un valor que nunca sería un dato válido (una calificación nunca es -1) y lo usas como señal de salida.
- El segundo if valida el rango. Si la calificación es inválida, mostramos un aviso y ejecutamos ``continue``: esta palabra salta todo lo que sigue en la vuelta actual y regresa directamente a evaluar la condición del while para la siguiente vuelta. La calificación inválida no se suma ni se cuenta. Fíjate en la diferencia: ``break`` abandona el ciclo por completo, ``continue`` solo abandona la vuelta actual.
- ``suma = suma + calif;`` acumula: toma la suma anterior, le agrega la nueva calificación y la guarda.
- ``cantidad = cantidad + 1;`` cuenta una calificación válida más.
- Después del ciclo: ``if (cantidad > 0)`` protege la división. Si el usuario escribió -1 inmediatamente sin capturar nada, cantidad sería 0 y dividir entre 0 tronaría el programa (o daría infinito con doubles). Esta verificación es un ejemplo de programación defensiva: anticipar el caso borde donde no hay datos.
- ``(double)suma / cantidad``: el ``(double)`` convierte suma a decimal ANTES de dividir. Sin esta conversión, ``suma / cantidad`` sería división entera: $85 / 2$ daría 42 en lugar de 42.5, porque al dividir dos int el resultado es int y se trunca. Este es un error clásico que ya debes dominar: cuando necesites decimales en una división, convierte al menos uno de los operandos a double.

Prueba mental: usuario escribe 80, 90, -1. Vuelta 1: calif=80, no es -1, es válida, suma=80, cantidad=1. Vuelta 2: calif=90, suma=170, cantidad=2. Vuelta 3: calif=-1, break. Promedio = $170.0 / 2 = 85$. Correcto.

Ejemplo 4: Juego de adivinar el número con bandera y switch expression

Problema: la computadora piensa un número del 1 al 20, el jugador tiene 5 intentos para adivinarlo. Después de cada intento, dar una pista con switch expression ("muy lejos", "cerca", etc. según la diferencia). Usar una bandera para saber si ganó.

```
Random azar = new Random();
int secreto = azar.Next(1, 21); // número aleatorio entre 1 y 20
int intentosMax = 5;
int intento = 1;
bool gano = false; // bandera: ¿ya adivinó?

Console.WriteLine("Adivina el número secreto (1-20). Tienes 5 intentos.");

while (intento <= intentosMax && !gano)
{
    Console.Write("Intento " + intento + ": ");
    int propuesta = int.Parse(Console.ReadLine());

    if (propuesta == secreto)
    {
        gano = true; // levanta la bandera: ¡ganó!
    }
    else
    {
        int diferencia = Math.Abs(propuesta - secreto);

        string pista = diferencia switch
        {
            <= 2 => "¡Muy caliente! Estás a nada.",
            <= 5 => "Caliente, te acercas.",
            <= 10 => "Tibio...",
            _ => "Frío, muy lejos."
        };

        Console.WriteLine(pista);

        if (propuesta < secreto)
        {
            Console.WriteLine("Pista extra: el número es MAYOR.");
        }
        else
        {
            Console.WriteLine("Pista extra: el número es MENOR.");
        }
    }
}
```

```

    }

    intento++;
}

if (gano)
{
    Console.WriteLine("¡Felicidades! Adivinaste en " + (intento - 1) + " intentos.");
}
else
{
    Console.WriteLine("Se acabaron los intentos. El número era " + secreto + ".");
}

```

Explicación línea por línea:

- ``Random azar = new Random();`` crea un generador de números aleatorios. ``azar.Next(1, 21)`` produce un entero entre 1 y 20 (el límite superior, 21, es exclusivo: nunca sale 21). Este es el número secreto.
- ``bool gano = false;`` inicializa la bandera en false: al inicio, nadie ha ganado.
- La condición del while combina dos cosas: ``intento <= intentosMax && !gano``. El ciclo sigue mientras queden intentos Y el jugador no haya ganado. En cuanto una de las dos falle, se detiene. Esta es la bandera controlando el ciclo, como vimos en la teoría.
- Si la propuesta es igual al secreto, levantamos la bandera: ``gano = true``. No hacemos break explícito: en la siguiente evaluación de la condición, ``!gano`` será false y el ciclo terminará solo. Es una forma limpia de salir.
- Si no adivinó, calculamos ``Math.Abs(propuesta - secreto)``: la diferencia absoluta (siempre positiva) entre la propuesta y el secreto. `Math.Abs` quita el signo negativo si lo hay.
- La switch expression convierte la diferencia en una pista de texto: ``<= 2`` es patrón relacional ("menor o igual que 2"), y ``_`` atrapa todo lo demás. Fíjate qué compacto quedó comparado con un if-else if-else equivalente.
- La pista extra de MAYOR/MENOR usa un if-else simple: si la propuesta es menor que el secreto, el secreto es mayor, y viceversa.
- ``intento++`` al final de cada vuelta avanza el contador de intentos. Ojo: va DESPUÉS de procesar la propuesta, y se ejecuta en todas las vueltas (esté dentro del if o del else), por eso está fuera de ambos, al final del bloque del while.

- Al terminar el ciclo, la bandera decide el mensaje final: si gano es true, felicitamos y mostramos en cuántos intentos lo logró (`intento - 1` porque intento ya se incrementó una vez de más al salir). Si es false, revelamos el número secreto.

Este ejemplo junta casi todo lo de la unidad: while con condición compuesta, bandera, if-else, switch expression con patrones relacionales, contadores y Math. Si entiendes este programa completo, dominas la unidad.

Ejemplo 5: Validador de contraseña con do-while

Problema: pedir una contraseña hasta que cumpla los requisitos (mínimo 8 caracteres, al menos un número) o hasta agotar 3 intentos. Combina do-while con validaciones múltiples.

```
int intentosRestantes = 3;
bool aceptada = false;
string password = "";

do
{
    Console.Write("Crea tu contraseña (mínimo 8 caracteres y al menos un número): ");
    password = Console.ReadLine();

    bool largoOk = password.Length >= 8;
    bool tieneNumero = false;

    int i = 0;
    while (i < password.Length && !tieneNumero)
    {
        if (char.IsDigit(password[i]))
        {
            tieneNumero = true;
        }
        i++;
    }

    if (largoOk && tieneNumero)
    {
        aceptada = true;
        Console.WriteLine("Contraseña aceptada. ¡Bienvenido!");
    }
    else
    {
        intentosRestantes--;

        if (!largoOk)
```

```

        {
            Console.WriteLine("Error: debe tener al menos 8 caracteres.");
        }
        if (!tieneNumero)
        {
            Console.WriteLine("Error: debe incluir al menos un número.");
        }

        if (intentosRestantes > 0)
        {
            Console.WriteLine("Te quedan " + intentosRestantes + " intentos.");
        }
    }
} while (!aceptada && intentosRestantes > 0);

if (!aceptada)
{
    Console.WriteLine("Demasiados intentos fallidos. Inténtalo más tarde.");
}

```

Explicación línea por línea:

- Declaramos `intentosRestantes` (contador descendente), `aceptada` (bandera) y `password` (vacía inicialmente).
- El do-while garantiza que se pida la contraseña al menos una vez. La condición de repetición: seguir mientras NO haya sido aceptada Y queden intentos.
- `password.Length >= 8` verifica la longitud. Length en un string dice cuántos caracteres tiene.
- Para buscar un número dentro del texto usamos un while que recorre cada carácter: `password[i]` es el carácter en la posición i (las posiciones empiezan en 0). `char.IsDigit(...)` pregunta si ese carácter es un dígito del 0 al 9. Es un ciclo de búsqueda con bandera: en cuanto encuentra un dígito, `tieneNumero` se vuelve true y la condición `!tieneNumero` detiene el ciclo sin revisar el resto. Eficiente y elegante.
- Si ambas validaciones pasan, levantamos la bandera `aceptada` y felicitamos.

- Si algo falla, decrementamos los intentos y mostramos mensajes específicos para cada problema. Fíjate que usamos dos ifs SEPARADOS (no else if) para los mensajes de error: queremos mostrar TODOS los problemas a la vez, no solo el primero. Si la contraseña es corta Y no tiene número, el usuario debe saber ambas cosas para corregirla de una vez. Esta es una decisión de diseño consciente: else if muestra un solo mensaje, ifs separados muestran todos.
- Al salir del ciclo, si la bandera nunca se levantó, informamos el bloqueo.

Este ejemplo muestra un patrón profesional real: validación de entrada con retroalimentación específica y límite de intentos. Lo verás en formularios web, apps bancarias y sistemas corporativos. La estructura do-while + bandera + contador descendente es una receta que puedes reutilizar en docenas de situaciones.

Práctica

Es hora de ensuciarte las manos. Los ejercicios están ordenados de menor a mayor dificultad. Intenta resolver cada uno POR TU CUENTA antes de mirar la solución: la programación se aprende programando, no leyendo. Si te atorras más de 20 minutos, mira la solución, entiéndela línea por línea, ciérrala y vuelve a escribirla tú solo. Ese es el ciclo de aprendizaje que funciona.

Ejercicios

1. Semáforo inteligente: pide al usuario el color del semáforo ("verde", "amarillo" o "rojo") y muestra qué debe hacer el conductor ("Avanza", "Reduce la velocidad" o "Detente"). Si escribe otro color, muestra "Color no válido". Usa switch. Normaliza la entrada a minúsculas para que "Verde" y "VERDE" también funcionen.
2. Calculadora de descuentos: una tienda da descuentos según el monto de compra: 20% si la compra es de \$1000 o más, 10% si es de \$500 o más, 5% si es de \$100 o más, y 0% si es menor. Pide el monto, calcula el descuento con if-else if-else (cuidado con el orden) y muestra el monto original, el descuento aplicado y el total a pagar.
3. Tabla de multiplicar con while: pide un número del 1 al 10 y muestra su tabla de multiplicar del 1 al 10 usando un ciclo while. Ejemplo: si el usuario escribe 7, muestra "7 x 1 = 7", "7 x 2 = 14", etc. Valida que el número esté entre 1 y 10; si no, muestra un mensaje de error y no muestres la tabla.
4. Suma hasta el límite: pide números al usuario uno por uno y ve acumulando la suma. El ciclo termina cuando la suma acumulada llegue o supere 100. Al final muestra cuántos

números se capturaron y la suma final. Usa un acumulador, un contador y un while. Pista: la condición del while depende del acumulador, no de un contador fijo.

5. Adivina con pistas de temperatura: modifica el juego de adivinar el número del Ejemplo 4 para que use un número secreto entre 1 y 50, dé 6 intentos, y las pistas sean con switch expression según la diferencia: diferencia 0 (ya adivinó), menor o igual a 3 "Hirviendo", menor o igual a 8 "Caliente", menor o igual a 15 "Tibio", lo demás "Frío". Además, lleva la cuenta de cuántas veces el jugador estuvo "Hirviendo" o "Caliente" usando un contador, y muéstralo al final como dato curioso.
6. Cajero automático simplificado: el usuario tiene un saldo inicial de \$5000. Muestra un menú con do-while y switch: 1) Consultar saldo, 2) Retirar (pide el monto; no puede retirar más del saldo ni cantidades negativas o cero), 3) Depositar (pide el monto; debe ser positivo), 0) Salir. Después de cada operación muestra el nuevo saldo. Si intenta retirar de más, muestra un error y no modifies el saldo. El menú se repite hasta elegir 0.
7. Números primos del 1 al 50 (reto): muestra todos los números primos entre 1 y 50. Un número es primo si solo es divisible entre 1 y entre sí mismo. Pista: usa un while externo que recorra los números del 2 al 50, y por cada número usa otro while (o un contador de divisores) que pruebe divisores desde 2 hasta la raíz del número o hasta numero/2. Si ningún divisor lo divide exactamente (residuo 0 con el operador %), entonces es primo. Este ejercicio combina ciclos anidados, el operador módulo y una bandera o contador.
8. Patrón de asteriscos (reto visual): pide un número n y dibuja un triángulo de asteriscos con n filas usando while anidados: la fila 1 tiene 1 asterisco, la fila 2 tiene 2, etc. Ejemplo con n=4:

* *** **** Pista: el ciclo externo controla la fila (de 1 a n) y el ciclo interno imprime los asteriscos de esa fila con Write (sin salto de línea), y al terminar cada fila un WriteLine vacío hace el salto.

Soluciones

Solución 1: Semáforo inteligente

```

Console.Write("Color del semáforo (verde/amarillo/rojo): ");
string color = Console.ReadLine().ToLower(); // normaliza a minúsculas

switch (color)
{
    case "verde":
        Console.WriteLine("Avanza.");
        break;
    case "amarillo":
        Console.WriteLine("Reduce la velocidad.");
        break;
    case "rojo":
        Console.WriteLine("Detente.");
        break;
    default:
        Console.WriteLine("Color no válido.");
        break;
}

```

Explicación: el truco está en `.ToLower()`, que convierte todo lo que escribió el usuario a minúsculas antes de comparar. Así "VERDE", "Verde" y "verde" caen en el mismo case. El default atrapa cualquier otra cosa ("azul", "morado", texto vacío). Cada case termina con break como exige C#. Este es el uso clásico del switch: una variable comparada contra valores constantes.

Solución 2: Calculadora de descuentos

```

Console.Write("Monto de la compra: $");
double monto = double.Parse(Console.ReadLine());

double porcentaje;

if (monto >= 1000)
{
    porcentaje = 0.20;
}
else if (monto >= 500)
{
    porcentaje = 0.10;
}
else if (monto >= 100)
{
    porcentaje = 0.05;
}
else
{
    porcentaje = 0.0;
}

double descuento = monto * porcentaje;
double total = monto - descuento;

Console.WriteLine("Monto original: $" + monto);
Console.WriteLine("Descuento (" + (porcentaje * 100) + "%): $" + descuento);
Console.WriteLine("Total a pagar: $" + total);

```

Explicación: el orden descendente es crítico aquí. Si `monto >= 100` estuviera primero, una compra de \$2000 obtendría solo 5% de descuento en lugar de 20%, porque 2000 también es >= 100 y caería en la primera rama verdadera. Siempre verifica: ¿esta condición atraparía casos que pertenecen a una rama posterior? Si sí, está mal ordenada. Guardamos el porcentaje en una variable y calculamos después: esto separa la DECISIÓN (qué descuento toca) del CÁLCULO (cuánto es en dinero), lo que hace el código más claro. Nota que usamos double para el monto porque el dinero tiene centavos.

Solución 3: Tabla de multiplicar con while

```

Console.Write("Escribe un número del 1 al 10: ");
int numero = int.Parse(Console.ReadLine());

if (numero < 1 || numero > 10)
{
    Console.WriteLine("Error: el número debe estar entre 1 y 10.");
}
else
{
    Console.WriteLine("Tabla del " + numero + ":");

    int i = 1;
    while (i <= 10)
    {
        int resultado = numero * i;
        Console.WriteLine(numero + " x " + i + " = " + resultado);
        i++;
    }
}

```

Explicación: primero validamos (programación defensiva: no tiene sentido calcular la tabla de un número fuera de rango). Solo si pasa la validación entramos al else donde vive el ciclo. El while usa un contador `i` que va de 1 a 10; en cada vuelta calcula y muestra la multiplicación. El incremento `i++` al final de cada vuelta es lo que evita el ciclo infinito. Fíjate en el patrón: validación con if afuera, trabajo con while adentro. Lo verás una y otra vez en código real.

Solución 4: Suma hasta el límite

```

int suma = 0;          // acumulador
int cantidad = 0;     // contador

Console.WriteLine("Escribe números. El ciclo termina cuando la suma llegue a 100.");

while (suma < 100)
{
    Console.Write("Número: ");
    int n = int.Parse(Console.ReadLine());

    suma = suma + n;      // acumulamos el valor (puede ser cualquiera)
    cantidad = cantidad + 1; // contamos uno más

    Console.WriteLine("Suma actual: " + suma);
}

Console.WriteLine("¡Llegaste a 100!");
Console.WriteLine("Capturaste " + cantidad + " números.");
Console.WriteLine("Suma final: " + suma);

```

Explicación: aquí la condición del while depende del ACUMULADOR (`suma < 100`), no de un contador fijo. No sabemos de antemano cuántas vueltas dará el ciclo: si el usuario escribe 100 de una vez, será una sola vuelta; si escribe de 1 en 1, serán 100 vueltas. Por eso no podemos usar un for clásico con límite fijo (lo verás en la Unidad 4): el while es la herramienta correcta cuando no sabes cuántas repeticiones habrá. El acumulador crece en cada vuelta, acercándose a la condición de salida: esto garantiza que el ciclo termine (siempre que el usuario no escriba solo ceros o negativos eternamente... un detalle que en un sistema real validarías, pero para el ejercicio está bien).

Solución 5: Adivina con pistas de temperatura

```

Random azar = new Random();
int secreto = azar.Next(1, 51); // entre 1 y 50
int intentosMax = 6;
int intento = 1;
bool gano = false;
int vecesCerca = 0; // contador de pistas "Hirviendo" o "Caliente"

Console.WriteLine("Adivina el número (1-50). Tienes 6 intentos.");

while (intento <= intentosMax && !gano)
{

```

```

Console.Write("Intento " + intento + ": ");
int propuesta = int.Parse(Console.ReadLine());

if (propuesta == secreto)
{
    gano = true;
}
else
{
    int diferencia = Math.Abs(propuesta - secreto);

    string pista = diferencia switch
    {
        <= 3 => "Hirviendo",
        <= 8 => "Caliente",
        <= 15 => "Tibio",
        _ => "Frío"
    };

    Console.WriteLine(pista + "...");

    if (diferencia <= 8)
    {
        vecesCerca++; // contador: estuvo cerca
    }

    if (propuesta < secreto)
    {
        Console.WriteLine("El número es MAYOR.");
    }
    else
    {
        Console.WriteLine("El número es MENOR.");
    }
}

intento++;
}

if (gano)
{
    Console.WriteLine("¡Ganaste en " + (intento - 1) + " intentos!");
}
else
{

```

```

        Console.WriteLine("Perdiste. El número era " + secreto + ".");
    }

    Console.WriteLine("Dato curioso: estuviste cerca " + vecesCerca + " veces.");

```

Explicación: es una evolución del Ejemplo 4 con tres cambios: rango más amplio (1-50), 6 intentos, y un contador `vecesCerca` que se incrementa cada vez que la diferencia es de 8 o menos (las categorías "Hirviendo" y "Caliente"). Fíjate cómo el contador vive DENTRO del else (solo cuenta cuando no adivinó) pero FUERA de la switch expression. La condición `diferencia <= 8` agrupa las dos categorías cercanas sin repetir la lógica de la pista. Al final mostramos el dato curioso siempre, gane o pierda: es información adicional que no afecta el juego. Este ejercicio practica algo muy real: agregar una métrica nueva a un programa existente sin romper lo que ya funcionaba.

Solución 6: Cajero automático simplificado

```

double saldo = 5000;
int opcion;

do
{
    Console.WriteLine("==== CAJERO AUTOMÁTICO =====");
    Console.WriteLine("1. Consultar saldo");
    Console.WriteLine("2. Retirar");
    Console.WriteLine("3. Depositar");
    Console.WriteLine("0. Salir");
    Console.Write("Elige: ");
    opcion = int.Parse(Console.ReadLine());
    Console.WriteLine();

    switch (opcion)
    {
        case 1:
            Console.WriteLine("Tu saldo es: $" + saldo);
            break;

        case 2:
            Console.Write("¿Cuánto quieres retirar? $");
            double retiro = double.Parse(Console.ReadLine());

            if (retiro <= 0)
            {
                Console.WriteLine("Error: el monto debe ser mayor que cero.");
            }

```

```

        else if (retiro > saldo)
        {
            Console.WriteLine("Error: saldo insuficiente. Tienes $" + saldo +
".");
        }
        else
        {
            saldo = saldo - retiro;
            Console.WriteLine("Retiro exitoso. Nuevo saldo: $" + saldo);
        }
        break;

    case 3:
        Console.Write("¿Cuánto quieres depositar? $");
        double deposito = double.Parse(Console.ReadLine());

        if (deposito <= 0)
        {
            Console.WriteLine("Error: el monto debe ser mayor que cero.");
        }
        else
        {
            saldo = saldo + deposito;
            Console.WriteLine("Depósito exitoso. Nuevo saldo: $" + saldo);
        }
        break;

    case 0:
        Console.WriteLine("Gracias por usar el cajero. ¡Hasta luego!");
        break;

    default:
        Console.WriteLine("Opción no válida.");
        break;
}

Console.WriteLine();

} while (opcion != 0);

```

Explicación: este es el ejercicio integrador de la unidad: combina do-while (menú), switch (opciones), if-else if-else (validaciones de montos) y una variable de estado (`saldo`) que persiste entre vueltas del ciclo. Puntos clave: el saldo se declara ANTES del do-while para que no se reinicie en cada vuelta (si lo declararas dentro, cada vez que el menú se repitiera el saldo volvería a \$5000: un error clásico de ámbito). En el retiro hay DOS validaciones en

orden: primero montos inválidos (≤ 0), luego saldo insuficiente. El orden importa: si alguien escribe -500, debe recibir "monto debe ser mayor que cero", no "saldo insuficiente". Cada validación protege la integridad del saldo: solo se modifica cuando todo está bien. Este patrón de "validar todo antes de modificar el estado" es fundamental en sistemas reales (bancos, inventarios, reservas): nunca dejes los datos a medias por una validación que llegó tarde.

Solución 7: Números primos del 1 al 50 (reto)

```
Console.WriteLine("Números primos del 1 al 50:");

int numero = 2;

while (numero <= 50)
{
    bool esPrimo = true; // bandera: asumimos que es primo hasta demostrar lo contrario
    int divisor = 2;

    while (divisor * divisor <= numero)
    {
        if (numero % divisor == 0)
        {
            esPrimo = false; // encontramos un divisor: no es primo
            break;           // ya no hace falta seguir buscando
        }
        divisor++;
    }

    if (esPrimo)
    {
        Console.Write(numero + " ");
    }

    numero++;
}

Console.WriteLine();
```

Explicación: hay dos ciclos anidados. El externo recorre los candidatos del 2 al 50. El interno busca divisores: prueba dividir el número entre 2, 3, 4... Si algún divisor lo divide exactamente ($\text{numero \% divisor} == 0$, el módulo da el residuo de la división), entonces no es primo: levantamos la bandera en false y rompemos el ciclo interno con break, porque ya no

necesitamos seguir buscando. El truco `divisor * divisor <= numero` es una optimización matemática: si un número tiene un divisor mayor que su raíz cuadrada, forzosamente tiene otro menor que la raíz, así que basta buscar hasta la raíz. Sin esta optimización el programa también funcionaría (probando hasta `numero - 1`), solo sería más lento. La bandera `esPrimo` se REINICIA en `true` para cada número nuevo: si la declararas fuera del ciclo externo, una vez que se volviera `false` se quedaría `false` para siempre y ningún número posterior se marcaría como primo. Este es un error clásico con banderas en ciclos anidados: la bandera debe reiniciarse en cada vuelta del ciclo externo. Al final, los primos se imprimen en la misma línea separados por espacios.

Solución 8: Patrón de asteriscos (reto visual)

```
Console.Write("¿Cuántas filas? ");
int n = int.Parse(Console.ReadLine());

int fila = 1;

while (fila <= n)
{
    int col = 1;

    while (col <= fila)
    {
        Console.Write("*");
        col++;
    }

    Console.WriteLine(); // salto de línea al terminar la fila
    fila++;
}
```

Explicación: el ciclo externo controla las filas (de 1 a `n`) y el interno dibuja cada fila. La clave está en la condición del ciclo interno: `col <= fila`. En la fila 1, el interno da 1 vuelta (un asterisco); en la fila 2, da 2 vueltas; y así sucesivamente. El ciclo interno usa `Write` (sin salto) para que los asteriscos queden pegados en el mismo renglón, y cuando termina, el `WriteLine` vacío del ciclo externo hace el salto a la siguiente fila. Fíjate que `col` se declara y se inicializa en 1 DENTRO del ciclo externo: debe reiniciarse en cada fila, igual que la bandera del ejercicio anterior. Si la declararas fuera, después de la primera fila `col` ya valdría más que `fila` y las siguientes filas saldrían vacías. Los ciclos anidados donde el interno depende de la variable del externo son la base para dibujar patrones, recorrer tablas y matrices (que verás en unidades posteriores).

Resumen de la unidad

- El ``if`` ejecuta un bloque solo si su condición es verdadera; el ``else`` define el camino alternativo cuando es falsa; el ``else if`` encadena múltiples condiciones que se evalúan en orden hasta que una sea verdadera.
- Los operadores de comparación (``==``, ``!=``, ``>``, ``<``, ``>=``, ``<=``) devuelven true o false; no confundas ``=`` (asignación) con ``==`` (comparación).
- Los operadores lógicos combinan condiciones: ``&&`` (Y, ambas deben ser true), ``||`` (O, basta una true), ``!`` (NO, invierte el valor); aprovecha el cortocircuito y usa paréntesis al mezclar ``&&`` con ``||``.
- Los ifs anidados modelan decisiones dependientes, pero más de dos niveles indican que debes reestructurar con early returns o condiciones planas.
- El operador ternario ``condicion ? valorSiTrue : valorSiFalse`` sirve para asignaciones simples de una línea; no lo anides.
- El ``switch`` dirige según el valor de una expresión entre múltiples ``case``; cada case lleva ``break``, y el ``default`` atrapa lo inesperado; úsalo cuando compares una sola variable contra valores constantes.
- El switch moderno soporta patrones: ``when`` agrega condiciones a un case, los type patterns verifican tipos, y los patrones relacionales (``< 13``) comparan rangos directamente.
- Las switch expressions (``variable switch { patron => valor, _ => otroValor }``) producen un valor de forma compacta; el compilador exige que sean exhaustivas.
- El ``while`` repite un bloque mientras su condición sea verdadera, evaluándola ANTES de cada vuelta (puede no ejecutarse nunca); el ``do-while`` evalúa al final y garantiza al menos una ejecución (no olvides su punto y coma).
- Todo ciclo debe tener algo que acerque su condición a terminarse; si no, es un ciclo infinito: el error más común y peligroso con while.
- Los tres patrones clásicos: contador (inicia en 0, suma 1, responde "¿cuántos?"), acumulador (inicia en 0 para sumas o 1 para multiplicaciones, suma valores variables, responde "¿cuánto en total?") y bandera (bool que inicia en false, marca si algo ocurrió, controla salidas de búsqueda).
- ``break`` abandona el ciclo por completo; ``continue`` salta a la siguiente vuelta; el valor centinela (-1, 0, texto vacío) es la señal clásica de "ya terminé" en ciclos con salida interna.

- Buenas prácticas: condiciones que se leen como frases, variables booleanas con buenos nombres para lógica compleja, validación de entrada antes de procesar, else/default como red de seguridad, y nombres descriptivos para contadores, acumuladores y banderas.

UNIDAD 4: Ciclos for/foreach y métodos

En la unidad anterior aprendiste a tomar decisiones y a repetir código con while y do-while. Esos ciclos son poderosos, pero tienen un detalle incómodo: cuando sabes de antemano cuántas veces quieres repetir algo, el while te obliga a manejar el contador a mano (declararlo, compararlo, incrementarlo en tres lugares distintos del código). Funciona, pero es fácil equivocarse: olvidas el incremento y tienes un ciclo infinito, o pones mal la condición y el ciclo da una vuelta de más o una de menos. Para esos casos tan comunes existe una herramienta diseñada exactamente para eso: el ciclo for. Y cuando lo que quieres es recorrer una colección de datos elemento por elemento (una lista de nombres, las calificaciones de un grupo), existe una herramienta aún más cómoda: el foreach.

Pero la segunda mitad de esta unidad es todavía más importante, porque marca el momento en que dejas de escribir "programitas" y empiezas a escribir software de verdad: los métodos. Un método es un bloque de código con nombre que hace una tarea específica y que puedes llamar (usar) cuantas veces quieras. Hasta ahora, todo tu código vivía dentro del Main, creciendo como una sola masa. Con los métodos vas a aprender a partir un problema grande en piezas pequeñas, cada una con su nombre y su responsabilidad. Es como cocinar: en lugar de hacer todo al mismo tiempo en un caos, tienes ayudantes (los métodos) que cada uno se encarga de algo: uno pica la verdura, otro prepara la salsa, otro pone la mesa. Tú solo coordinas.

¿Por qué importa esto en la vida real? Porque ningún programa profesional es una lista gigante de instrucciones. Las apps que usas todos los días tienen miles o millones de líneas organizadas en métodos (y más adelante, en clases y objetos). Cuando tu banco calcula los intereses de tu cuenta, llama a un método. Cuando un videojuego dibuja a cada enemigo en pantalla, usa un ciclo que recorre la lista de enemigos y un método que dibuja a cada uno. Los ciclos for y foreach procesan datos en masa: tu lista de contactos, los productos de una tienda, los mensajes de tu bandeja. Y los métodos permiten que equipos enteros de programadores trabajen juntos: cada quien escribe sus métodos, y todos los usan. Sin métodos, el código se vuelve un monstruo imposible de mantener: la misma lógica copiada y pegada en diez lugares, y cuando hay que corregir un error hay que corregirlo diez veces.

Al terminar esta unidad podrás escribir programas organizados como un profesional: recorrer arreglos y listas con for y foreach eligiendo el ciclo correcto para cada situación, dividir tus programas en métodos con nombres claros que reciben parámetros y devuelven resultados, usar ref y out cuando necesites que un método modifique variables del que lo llama, crear varias versiones del mismo método con sobrecarga, y simplificar llamadas con parámetros opcionales. Vas a construir programas como un menú de restaurante completo donde cada

opción es un método, o un sistema de calificaciones donde cada cálculo vive en su propio método reutilizable. Tu código se va a ver más corto, más limpio y mucho más fácil de entender.

Esta unidad cubre tres clases: la clase 10 (el ciclo for clásico, el foreach para colecciones y arreglos, cuándo usar cada uno, y las instrucciones break y continue), la clase 11 (métodos: cómo declararlos y llamarlos, parámetros, valores de retorno, ámbito de variables y métodos void) y la clase 12 (parámetros ref, out e in, sobrecarga de métodos y parámetros opcionales). Presta atención especial a los errores clásicos de esta unidad: modificar una colección mientras la recorres con foreach (el programa truena), usar ref con una variable sin inicializar (el compilador no te deja), y olvidar asignar un parámetro out dentro del método (tampoco compila). Son errores que el compilador detecta, lo cual es una buena noticia: mejor un error al compilar que un programa que falla a la mitad.

Explicación detallada

El ciclo for: repetir un número exacto de veces

Imagina que tienes que repartir volantes a 50 casas. No necesitas pensar "¿ya terminé?" después de cada casa contando de forma creativa: sabes que son 50 y punto. Cuentas del 1 al 50 y cuando llegas al 50, terminas. El ciclo for es exactamente eso: un ciclo diseñado para cuando sabes cuántas repeticiones necesitas (o puedes calcularlas antes de empezar).

Su sintaxis junta en un solo lugar las tres cosas que en el while andaban separadas:

```
for (inicialización; condición; actualización)
{
    // bloque que se repite
}
```

Mira un ejemplo concreto y luego lo desmenuzamos:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine("Vuelta número " + i);
}
```

Esto muestra: Vuelta número 0 Vuelta número 1 Vuelta número 2 Vuelta número 3 Vuelta número 4

Analicemos cada parte del paréntesis, separadas por punto y coma:

1. Inicialización: ``int i = 0``. Se ejecuta UNA sola vez, antes de empezar el ciclo. Aquí declaras y das valor inicial a la variable de control (casi siempre llamada ``i``, de "índice"). Esta variable solo existe dentro del for: cuando el ciclo termina, desaparece.
2. Condición: ``i < 5``. Se evalúa ANTES de cada vuelta, igual que en el while. Mientras sea verdadera, el ciclo continúa. En cuanto sea falsa, el ciclo termina.
3. Actualización: ``i++``. Se ejecuta AL FINAL de cada vuelta, después del bloque. Normalmente incrementa la variable de control en 1.

El flujo completo es: inicializa -> ¿condición verdadera? -> ejecuta bloque -> actualiza -> ¿condición verdadera? -> ejecuta bloque -> actualiza... hasta que la condición sea falsa.

Sigamos la ejecución del ejemplo: `i` empieza en 0. ¿0 < 5? Sí: muestra "Vuelta número 0", `i` pasa a 1. ¿1 < 5? Sí: muestra "Vuelta número 1", `i` pasa a 2... Así hasta que `i` vale 5: ¿5 < 5? No. El ciclo termina. Total: 5 vueltas, con `i` tomando los valores 0, 1, 2, 3, 4.

Fíjate en un detalle que confunde a muchos principiantes: el ciclo da 5 vueltas pero `i` nunca vale 5 dentro del bloque. La condición ``i < 5`` con `i` empezando en 0 produce exactamente 5 vueltas (0 a 4). Esta combinación (empezar en 0, condición con ``<``) es EL patrón estándar en programación, y hay una razón poderosa: los arreglos (que verás en un momento) empiezan en la posición 0, así que este patrón recorre un arreglo de 5 elementos visitando las posiciones 0, 1, 2, 3, 4 sin salirse. Memorízalo: ``for (int i = 0; i < n; i++)`` da exactamente `n` vueltas. Es el ciclo más escrito en la historia de la programación.

Puedes variar cada parte según necesites. Contar de 1 a 10:

```
for (int i = 1; i <= 10; i++)
{
    Console.Write(i + " ");
}
// muestra: 1 2 3 4 5 6 7 8 9 10
```

Contar hacia atrás de 10 a 1 (útil para cuentas regresivas):

```
for (int i = 10; i >= 1; i--)
{
    Console.Write(i + " ");
}
Console.WriteLine("¡Despegue!");
```

Avanzar de 2 en 2 (útil para números pares):

```
for (int i = 2; i <= 20; i += 2)
{
    Console.Write(i + " ");
}
// muestra: 2 4 6 8 10 12 14 16 18 20
```

La actualización no tiene que ser `i++`: puede ser `i--`, `i += 2`, `i \*= 2` o lo que necesites. La inicialización no tiene que empezar en 0. La condición no tiene que usar `<`. El for es flexible, pero el patrón estándar `for (int i = 0; i < n; i++)` debe ser tu punto de partida siempre que no tengas una razón para hacer otra cosa.

while vs for: ¿cuál usar?

Ya conoces dos ciclos que repiten código. ¿Cuándo usas cada uno? La regla es simple y práctica:

- Usa `for` cuando sabes (o puedes calcular) cuántas vueltas darás ANTES de empezar el ciclo. Ejemplos: "repite 10 veces", "recorre las 30 calificaciones del grupo", "muestra la tabla del 1 al 10". El for junta inicialización, condición y actualización en un solo lugar, lo que hace imposible olvidar el incremento (adiós, ciclos infinitos por descuido).
- Usa `while` cuando NO sabes cuántas vueltas serán y la condición de salida depende de algo que pasa DENTRO del ciclo. Ejemplos: "pide la contraseña hasta que sea correcta", "lee números hasta que el usuario escriba 0", "sigue jugando mientras el jugador quiera". Aquí el for quedaría forzado y raro.
- Usa `do-while` cuando el bloque debe ejecutarse al menos una vez sí o sí (menús, primera petición de datos).

Mira la diferencia con un ejemplo que ya conoces: mostrar los números del 1 al 100. Con while:

```
int n = 1;
while (n <= 100)
{
    Console.WriteLine(n);
    n++;
}
```

Con for:


```
for (int i = 1; i <= 100; i++)
{
    Console.WriteLine(i);
}
```

El for es más compacto y más seguro: la variable de control, su límite y su incremento están juntos y a la vista. Con el while, el `n++` está escondido al final del bloque, lejos de la condición, y es fácil olvidarlo o modificarlo por accidente. Cuando el número de vueltas es conocido, el for gana siempre.

Un error clásico con el for es el famoso "error por uno" (off-by-one error): dar una vuelta de más o una de menos. Ocurre cuando mezclas mal el inicio y la condición: `for (int i = 0; i <= 5; i++)` da 6 vueltas (0 a 5), no 5. Si querías 5 vueltas, debía ser `i < 5` o empezar en 1 con `i <= 5`. La forma de no equivocarte: decide primero cuántas vueltas quieres (n), y usa el patrón estándar `for (int i = 0; i < n; i++)`. Si necesitas que i tome valores del 1 al n por alguna razón, usa `for (int i = 1; i <= n; i++)`. Elige un patrón y sé consistente: no mezcles inicios en 0 con condiciones `<=` porque ahí es donde nacen los errores por uno.

Arreglos: guardar muchos datos en una variable

Antes de ver el foreach necesitas conocer los arreglos, porque son su compañero inseparable. Un arreglo (array) es una variable que guarda VARIOS valores del mismo tipo, ordenados y numerados. Piensa en un edificio de departamentos: el edificio es el arreglo, cada departamento es una posición, y el número de departamento es el índice. Todos los departamentos del edificio son del mismo tipo (viviendas), igual que todos los elementos de un arreglo son del mismo tipo de dato.

```
int[] calificaciones = { 85, 92, 78, 95, 88 };
```

Esta línea crea un arreglo de 5 enteros. Los corchetes `[]` después del tipo dicen "esto no es un int, es un arreglo de int". Los valores van entre llaves, separados por comas. Cada valor ocupa una posición numerada empezando en CERO:

- calificaciones[0] vale 85
- calificaciones[1] vale 92
- calificaciones[2] vale 78
- calificaciones[3] vale 95
- calificaciones[4] vale 88

Que el primer índice sea 0 y no 1 es una de las cosas más contraintuitivas para los principiantes, pero tiene razones históricas y prácticas. Acéptalo como regla: en C# (y en casi todos los lenguajes), los arreglos empiezan en 0. Un arreglo de 5 elementos tiene índices del 0 al 4. Si intentas leer `calificaciones[5]`, el programa truena con un error `IndexOutOfRangeException` ("índice fuera de rango"): estás tocando un departamento que no existe en el edificio.

También puedes crear un arreglo vacío especificando su tamaño y llenarlo después:

```
string[] nombres = new string[3]; // arreglo de 3 strings, todos null al inicio
nombres[0] = "Ana";
nombres[1] = "Luis";
nombres[2] = "Marta";
```

La palabra `new` reserva el espacio en memoria. `new string[3]` crea 3 casillas. Al inicio, cada casilla tiene el valor por defecto del tipo: 0 para números, false para bool, null para strings y objetos. Luego asignas valores a cada posición con el índice entre corchetes.

Para saber cuántos elementos tiene un arreglo, usas su propiedad `Length`:

```
int[] calificaciones = { 85, 92, 78, 95, 88 };
Console.WriteLine("El arreglo tiene " + calificaciones.Length + " elementos."); //
5
```

`Length` es tu mejor amigo para recorrer arreglos sin salirte del rango: el último índice válido siempre es `Length - 1`. Por eso el patrón estándar del `for` encaja perfecto con arreglos:

```
int[] calificaciones = { 85, 92, 78, 95, 88 };

for (int i = 0; i < calificaciones.Length; i++)
{
    Console.WriteLine("Calificación " + (i + 1) + ": " + calificaciones[i]);
}
```

`i < calificaciones.Length` garantiza que `i` llegue máximo a 4 (`Length - 1`), justo el último índice válido. Este patrón (`for + Length`) es tan común que lo escribirás cientos de veces. Si mañana el arreglo crece a 100 elementos, el ciclo se adapta solo: no hay que cambiar ningún número mágico en el código. Esa es la diferencia entre código frágil (con el 5 escrito a mano) y código robusto (que pregunta el tamaño).

El foreach: recorrer sin preocuparte por índices

El patrón `for + Length` funciona, pero cuando lo único que quieres es VISITAR cada elemento del arreglo (leerlos uno por uno, sin modificarlos, sin necesitar el número de posición), hay una forma más simple y más segura: el `foreach`. Su filosofía es: "para cada elemento de esta colección, haz esto". Tú no manejas índices, no calculas límites, no te puedes salir del rango. Es imposible cometer un error por uno con `foreach`.

```
int[] calificaciones = { 85, 92, 78, 95, 88 };

foreach (int calif in calificaciones)
{
    Console.WriteLine("Calificación: " + calif);
}
```

Se lee casi como español: "para cada `calif` (entero) en `calificaciones`, muestra la calificación". En cada vuelta, la variable ``calif`` toma automáticamente el siguiente valor del arreglo: primero 85, luego 92, luego 78... hasta terminar. Cuando se acaban los elementos, el ciclo termina solo. No hay `i`, no hay `Length`, no hay ``++`` que olvidar.

La variable del `foreach` (``calif`` en el ejemplo) es de solo lectura dentro del ciclo: puedes LEER su valor pero no puedes asignarle otro (el compilador no te deja). Esto es deliberado: el `foreach` está diseñado para recorrer, no para modificar.

El `foreach` no solo sirve para arreglos: funciona con cualquier colección: listas (`List<T>`, que verás en detalle en otra unidad), strings (recorre sus caracteres uno por uno), y más. Mira un ejemplo con string, que también es una colección de caracteres:

```
string mensaje = "Hola";

foreach (char letra in mensaje)
{
    Console.WriteLine(letra);
}

// muestra H, o, l, a en líneas separadas
```

for vs foreach: ¿cuál usar?

La decisión entre `for` y `foreach` depende de lo que necesites hacer dentro del ciclo:

Usa ``foreach`` cuando:

- Solo necesitas LEER cada elemento, uno por uno.
- No te importa (o no necesitas) el número de posición.
- Quieres el código más simple y seguro posible.

Es el valor por defecto: si dudas, usa foreach.

Usa `for` cuando:

- Necesitas el ÍNDICE (la posición) del elemento. Ejemplo: mostrar "Alumno 1: Ana, Alumno 2: Luis" donde el número es la posición.
- Necesitas MODIFICAR los elementos del arreglo. Ejemplo: duplicar cada calificación. (El foreach no permite modificar la variable de recorrido.)
- Necesitas recorrer el arreglo en orden inverso, de 2 en 2, o solo una parte de él.
- Necesitas trabajar con DOS arreglos al mismo tiempo usando el mismo índice (por ejemplo, nombres[i] con calificaciones[i]).

Mira un ejemplo donde el for es obligatorio: modificar elementos.

```
int[] precios = { 100, 200, 300 };

// Aplicar 10% de descuento a cada precio
for (int i = 0; i < precios.Length; i++)
{
    precios[i] = precios[i] - (precios[i] * 10 / 100);
}

foreach (int p in precios)
{
    Console.WriteLine("Precio con descuento: $" + p);
}
```

El for modifica cada posición usando el índice; el foreach muestra los resultados. Cada uno en lo que es mejor. Con el tiempo, elegir entre for y foreach se vuelve instinto: ¿necesito la posición o modificar? for. ¿Solo leer cada elemento? foreach.

El error clásico del foreach: modificar la colección mientras la recorres

Hay un error tan común que merece su propia sección con letras grandes: NO puedes agregar ni quitar elementos de una colección mientras la recorres con foreach. El programa truena con un error `InvalidOperationException` ("la colección fue modificada"). Mira el ejemplo del crimen:

```
List<string> invitados = new List<string> { "Ana", "Luis", "Marta", "Pedro" };

foreach (string nombre in invitados)
{
    if (nombre == "Luis")
    {
        invitados.Remove(nombre); // ¡ERROR! No se puede modificar durante el
foreach
    }
}
```

¿Por qué es un error? Porque el foreach usa internamente un "enumerador" que lleva la cuenta de en qué elemento va. Si agregas o quitas elementos a la mitad, el enumerador se confunde: las posiciones cambian y ya no sabe dónde está. En lugar de dar resultados incorrectos en silencio (lo peor que puede hacer un programa), C# prefiere tronar con un error claro. Es una protección, no un castigo.

¿Cómo se resuelve? Si necesitas quitar elementos según una condición, recorre con un for CLÁSICO de atrás hacia adelante (del último al primero): al ir hacia atrás, quitar un elemento no afecta las posiciones de los que faltan por revisar.

```
for (int i = invitados.Count - 1; i >= 0; i--)
{
    if (invitados[i] == "Luis")
    {
        invitados.RemoveAt(i); // quitar hacia atrás sí es seguro
    }
}
```

Si necesitas agregar elementos, primero termina el foreach guardando lo que necesitas en otra lista, y después del ciclo haz las modificaciones.

La regla de oro: durante un foreach, la colección es de solo lectura en cuanto a su estructura (no agregar, no quitar). Leer y modificar las PROPIEDADES de los elementos (si son objetos) sí se puede, pero cambiar la lista misma, no.

break y continue: controlar el ciclo desde adentro

Ya los viste en la unidad 3, pero aquí los formalizamos porque con for y foreach se usan muchísimo:

- ``break``: abandona el ciclo COMPLETO inmediatamente, saltando a la primera línea después del ciclo. Se usa para "deja de buscar, ya lo encontré".

```
int[] numeros = { 3, 7, 12, 5, 9 };
int buscado = 12;

for (int i = 0; i < numeros.Length; i++)
{
    if (numeros[i] == buscado)
    {
        Console.WriteLine("Encontrado en la posición " + i);
        break; // ya lo encontramos, no sigas buscando
    }
}
```

Sin el break, el ciclo seguiría revisando el 5 y el 9 inútilmente. Con break, termina en cuanto cumple su misión. En búsquedas sobre miles de elementos, esto ahorra muchísimo trabajo.

- ``continue``: abandona solo la VUELTA ACTUAL y salta a la siguiente. Se usa para "este elemento no me interesa, pasa al siguiente".

```
for (int i = 1; i <= 10; i++)
{
    if (i % 2 == 0)
    {
        continue; // los pares se saltan
    }
    Console.Write(i + " "); // solo se muestran los impares
}
// muestra: 1 3 5 7 9
```

El ``continue`` es elegante para filtrar: en lugar de meter todo tu código dentro de un ``if (condición) { ... }`` gigante, pones al inicio ``if (!condición) continue;`` y el resto del código queda al nivel principal, sin anidación. Es otra forma del principio de "salida temprana" que viste en la unidad 3, aplicado a ciclos.

Una advertencia: no abuses de break y continue. Un ciclo con tres breaks y dos continues escondidos es difícil de seguir: nadie sabe por dónde va a salir. Úsalos para casos claros (búsqueda con break, filtro con continue al inicio) y evita convertir tus ciclos en laberintos de saltos.

Métodos: tus ayudantes personales

Hasta ahora todo tu código ha vivido dentro del método Main, creciendo línea tras línea. Para programas pequeños está bien, pero imagina un programa de 500 líneas todo en Main: encontrar algo sería como buscar un calcetín en una montaña de ropa. Los métodos resuelven esto partiendo el programa en piezas pequeñas, cada una con un nombre que dice qué hace y que puedes usar cuantas veces quieras.

Piensa en una cocina de restaurante. El chef no hace todo solo: tiene un ayudante que pica verduras, otro que prepara salsas, otro que hornea el pan. Cada ayudante sabe hacer SU tarea y el chef solo les dice qué hacer y cuándo. Los métodos son tus ayudantes: tú defines una vez cómo se hace una tarea (por ejemplo, "calcular el promedio"), le pones nombre, y luego la usas cada vez que la necesites sin repetir el código.

Un método tiene dos momentos en su vida: la DECLARACIÓN (donde defines qué hace, una sola vez) y la LLAMADA (donde lo usas, cuantas veces quieras). Mira la anatomía completa:

```
// DECLARACIÓN del método: define qué hace
static double CalcularPromedio(double a, double b, double c)
{
    double suma = a + b + c;
    double promedio = suma / 3;
    return promedio;
}

// ... en el Main:
// LLAMADA al método: lo usamos
double prom = CalcularPromedio(80, 90, 100);
Console.WriteLine("Promedio: " + prom); // 90
```

Desmenucemos la declaración parte por parte:

- ``static``: por ahora, todos tus métodos llevarán esta palabra. Significa que el método pertenece a la clase y no a un objeto específico. Cuando estudies programación orientada a objetos entenderás la diferencia a fondo; por ahora, solo recuerda: si tu método está en la misma clase que el Main y lo llamas desde el Main, necesita ``static``. Si lo olvidas, el compilador te dirá exactamente eso.
- ``double``: el tipo de dato que el método DEVUELVE (su resultado). Este método calcula un promedio que puede tener decimales, así que devuelve double.
- ``CalcularPromedio``: el nombre del método. Por convención en C#, los nombres de métodos empiezan con mayúscula y usan PascalCase (cada palabra con mayúscula).

inicial, sin espacios ni guiones bajos): `CalcularPromedio`, `MostrarMenu`, `ValidarPassword`. Y la buena práctica de oro: los métodos se nombran con VERBOS (acciones): `Calcular`, `Mostrar`, `Validar`, `Buscar`, `Guardar`. Un método hace algo, así que su nombre debe decir qué hace.

- ``(double a, double b, double c)``: los PARÁMETROS: los datos que el método necesita para trabajar. Son como los ingredientes que le das al ayudante de cocina. Cada parámetro tiene tipo y nombre. Este método necesita tres calificaciones.
- El cuerpo entre llaves: las instrucciones que ejecutan la tarea.
- ``return promedio``: devuelve el resultado al que llamó al método. El `return` hace dos cosas: entrega el valor Y termina el método inmediatamente (igual que el `early return` de la unidad 3).

En la llamada, ``CalcularPromedio(80, 90, 100)``: los valores 80, 90, 100 se llaman ARGUMENTOS (los valores reales que le pasas). El método los recibe en sus parámetros a, b, c, hace el cálculo y devuelve 90, que guardamos en la variable ``prom``. La distinción parámetro/argumento: parámetro es la variable en la declaración (a, b, c), argumento es el valor en la llamada (80, 90, 100). En la práctica la gente los mezcla, pero conocer la diferencia te hace sonar profesional.

La magia real: una vez declarado, puedes llamar al método cuantas veces quieras:

```
double prom1 = CalcularPromedio(80, 90, 100);  
double prom2 = CalcularPromedio(70, 75, 80);  
double prom3 = CalcularPromedio(95, 100, 98);
```

Tres líneas usan la misma lógica sin repetirla. Si mañana cambias la fórmula del promedio (por ejemplo, a 4 calificaciones), la cambias en UN solo lugar y las tres llamadas se benefician. Esto tiene un nombre elegante: el principio DRY (Don't Repeat Yourself, "no te repitas"). Cada vez que te descubras copiando y pegando el mismo bloque de código dos veces, es señal de que ese bloque debería ser un método.

Métodos void: cuando no hay nada que devolver

No todos los métodos calculan algo. Algunos solo HACEN algo: mostrar un menú, imprimir un encabezado, guardar un archivo. Esos métodos no devuelven ningún valor, y se declaran con la palabra ``void`` (vacío) en lugar de un tipo de retorno:


```
static void MostrarEncabezado(string titulo)
{
    Console.WriteLine("=====");
    Console.WriteLine("    " + titulo);
    Console.WriteLine("=====");
}

// Llamada:
MostrarEncabezado("MI TIENDA");
MostrarEncabezado("REPORTE DE VENTAS");
```

Un método void no lleva `return` con valor (aunque puede llevar un `return;` solo, sin valor, para terminar antes de tiempo: otra aplicación del early return). Y en la llamada, no lo asignas a ninguna variable: simplemente lo llamas como una instrucción.

¿Cómo decides si un método debe devolver algo o ser void? Pregúntate: "¿el que llama necesita un RESULTADO para seguir trabajando, o solo necesita que se HAGA algo?" Si necesita un resultado (un cálculo, una búsqueda, una validación que responde sí/no), devuelve un valor. Si solo necesita que se ejecute una acción (mostrar, guardar, enviar), usa void. Con el tiempo este instinto se afina solo.

Parámetros: cómo entra la información al método

Los parámetros son el puente entre el que llama y el método. Ya viste parámetros simples; ahora veamos los detalles que importan:

Los parámetros son COPIAS: cuando llamas a un método con una variable, lo que recibe el método es una copia del valor, no la variable original. Si el método modifica su parámetro, la variable original NO cambia. Esto se llama "paso por valor" y es el comportamiento por defecto en C#:

```
static void IntentarCambiar(int x)
{
    x = 999;
    Console.WriteLine("Dentro del método, x vale: " + x);
}

int numero = 5;
IntentarCambiar(numero);
Console.WriteLine("Fuera del método, numero vale: " + numero);
// Muestra: dentro 999, fuera 5. ¡La original no cambió!
```

Piensa en ello como una fotocopia: le das al ayudante una copia de tu receta. Si él raya su copia, tu receta original sigue intacta. Esto es una protección: los métodos no pueden romper tus variables por accidente. (En la clase 12 verás ref y out, que son la excepción deliberada a esta regla.)

Puedes tener tantos parámetros como necesites, separados por comas, cada uno con su tipo. Y en la llamada debes pasar el mismo número de argumentos, en el mismo orden y de tipos compatibles:

```
static double CalcularAreaRectangulo(double base_, double altura)
{
    return base_ * altura;
}

double area = CalcularAreaRectangulo(5.0, 3.0); // base_=5.0, altura=3.0
```

Ojo con el orden: en un método como `Restar(a, b)` el orden lo es todo. Un truco de C# moderno para no equivocarte: los argumentos nombrados.

```
double area = CalcularAreaRectangulo(base_: 5.0, altura: 3.0);
```

Con `nombreParametro: valor` dejas explícito qué valor va a cada parámetro. Útil cuando un método tiene muchos parámetros del mismo tipo.

Valores de retorno: cómo sale la información del método

El `return` entrega el resultado y termina el método. Algunas reglas importantes:

1. El valor del return debe ser compatible con el tipo de retorno declarado. Si el método dice `int`, no puedes devolver un string. El compilador lo verifica.
2. Todo camino del método debe devolver un valor (excepto en void). Si tienes un if-else, AMBAS ramas deben tener return (o una al final). El compilador te obliga: "no todas las rutas de acceso devuelven un valor".

```
static string ClasificarEdad(int edad)
{
    if (edad >= 18)
    {
        return "Adulto";
    }
    else
    {
        return "Menor";
    }
    // Sin el else con return, el compilador se quejaría.
}
```

3. Después del return, nada se ejecuta. Cualquier código después de un return en el mismo bloque es código muerto (el compilador te avisa).
4. Puedes devolver el resultado de una expresión directamente, sin variable intermedia. Para métodos cortos es más limpio; para cálculos con varios pasos, las variables intermedias con buenos nombres documentan el proceso.

Ámbito de variables: dónde vive cada variable

El ámbito (scope) de una variable es la región del código donde esa variable existe y puede usarse. Es como el territorio de cada variable: fuera de su territorio, no existe. Hay tres niveles que debes dominar:

1. Variables LOCALES: declaradas dentro de un método (o dentro de un bloque como un if o un for). Solo existen dentro de ese método o bloque. Los parámetros también son locales al método.

```
static void Ejemplo()
{
    int x = 10; // local al método Ejemplo

    if (x > 5)
    {
        int y = 20; // local al bloque if: solo existe entre estas llaves
        Console.WriteLine(x + y); // OK: x y y existen aquí
    }
    // Console.WriteLine(y); // ERROR: y ya no existe aquí
}
// Console.WriteLine(x); // ERROR: x no existe fuera del método
```

2. Variables de CICLO: la `i` del for solo existe dentro del for. Por eso puedes usar `i` en diez ciclos for distintos sin conflicto: cada una vive y muere en su ciclo.
3. Variables de CLASE (campos): declaradas fuera de los métodos, directamente en la clase. Existen para todos los métodos de la clase y viven durante toda la ejecución. Se usan para datos que varios métodos necesitan compartir (como el saldo del cajero que persiste entre opciones del menú).

```
class Programa
{
    static double saldo = 5000; // campo: visible para todos los métodos

    static void Retirar(double monto)
    {
        saldo = saldo - monto; // puede leer y modificar el campo
    }

    static void MostrarSaldo()
    {
        Console.WriteLine("Saldo: " + saldo); // también lo ve
    }
}
```

Los errores clásicos de ámbito: usar una variable fuera de su bloque (declara la variable ANTES del bloque, en el ámbito donde la necesitas) y el sombreado (declarar una local con el mismo nombre que un campo: compila pero confunde; evítalo con nombres distintos).

La regla de oro del ámbito: declara cada variable en el ámbito MÁS PEQUEÑO posible donde se necesite.

Métodos que se llaman entre sí: construyendo programas por capas

Los métodos no solo son llamados desde el Main: pueden llamarse entre ellos, formando capas. Un método de alto nivel coordina, llamando a métodos más específicos que hacen el trabajo detallado. Esto se llama descomposición y es la habilidad central del diseño de software:

```

static void Main()
{
    MostrarReporte(); // el Main solo coordina
}

static void MostrarReporte()
{
    MostrarEncabezado("REPORTE DE VENTAS");
    double total = CalcularTotalVentas();
    Console.WriteLine("Total: $" + total);
    MostrarPie();
}

static double CalcularTotalVentas()
{
    double[] ventas = { 1500, 2300, 1800, 3100 };
    double suma = 0;
    foreach (double v in ventas)
    {
        suma += v;
    }
    return suma;
}

static void MostrarPie()
{
    Console.WriteLine("Fin del reporte.");
}

```

¿Ves la estructura? Main llama a MostrarReporte. MostrarReporte llama a MostrarEncabezado, CalcularTotalVentas y MostrarPie. Cada método hace UNA cosa y la hace bien. Si quieres cambiar cómo se calcula el total, solo tocas CalcularTotalVentas. Esta organización es lo que permite que programas de miles de líneas sigan siendo manejables.

Hay un límite práctico que debes respetar: un método debe ser CORTO. La guía profesional dice que un método debería caber en tu pantalla sin hacer scroll (típicamente menos de 20-30 líneas) y hacer UNA sola cosa (principio de responsabilidad única). Si tu método hace tres cosas distintas, son tres métodos esperando a nacer. ¿Cómo saber si un método hace más de una cosa? Intenta describirlo en una frase: si necesitas usar la palabra "y" ("este método calcula el promedio Y lo muestra Y lo guarda"), está haciendo demasiado. Pártelo.

Parámetros ref: cuando el método SÍ debe modificar tu variable

Recuerda que por defecto los parámetros son copias: el método no puede modificar tu variable original. Pero a veces SÍ quieres que la modifique. Ejemplo clásico: un método que intercambia dos valores, o uno que actualiza un contador que vive fuera. Para eso existe `ref` (referencia): en lugar de pasar una copia, pasas la variable misma.

```
static void Duplicar(ref int numero)
{
    numero = numero * 2; // modifica la variable ORIGINAL
}

int valor = 10;
Duplicar(ref valor);
Console.WriteLine(valor); // 20: ¡la original cambió!
```

Piensa en la diferencia con una analogía: pasar por valor es darle al ayudante una FOTOCOPIA de tu receta (puede rayarla sin dañar la original). Pasar con ref es darle tu RECETA ORIGINAL: lo que haga en ella, queda hecho. Por eso ref es poderoso pero debe usarse con cuidado y solo cuando realmente necesitas modificar la variable del que llama.

La sintaxis tiene una regla estricta: la palabra `ref` debe aparecer en DOS lugares: en la declaración del método (`static void Duplicar(ref int numero)`) y en la llamada (`Duplicar(ref valor)`). Si falta en alguno de los dos, no compila. Esto es deliberado: quien lee la llamada ve el `ref` y sabe inmediatamente "cuidado, este método puede modificar mi variable". Es honestidad sintáctica.

Otra regla estricta: la variable que pasas con ref debe estar INICIALIZADA antes de la llamada. Tiene sentido: como el método va a leer y modificar esa variable, debe tener un valor válido desde el inicio.

```
int valor; // declarada pero SIN valor
Duplicar(ref valor); // ERROR de compilación: la variable no está inicializada
```

El compilador te protege: "si me vas a dar tu variable para que la modifique, más te vale que tenga un valor inicial". Este es uno de los errores clásicos de la unidad: olvidar inicializar antes de pasar con ref. La solución es trivial (dale un valor inicial), pero el error confunde a los principiantes porque el mensaje del compilador habla de "variable no asignada".

¿Cuándo usar ref en la vida real? Los casos legítimos son pocos pero importantes: métodos que deben devolver MÁS de un valor modificado (aunque para eso hoy se prefiere out o

tuplas), algoritmos que transforman variables en su lugar (como ordenar un arreglo pasándolo con ref, aunque los arreglos ya son referencias por naturaleza), e interoperabilidad con código antiguo. La guía profesional: usa ref solo cuando el método necesita LEER el valor actual de la variable Y ADEMÁS modificarlo. Si solo necesita modificarlo (sin importar su valor previo), `out` es la herramienta correcta.

Parámetros out: cuando el método debe PRODUCIR un valor

`out` es el hermano de ref, con una diferencia clave: el método NO lee el valor inicial de la variable, solo está obligado a ASIGNARLE un valor antes de terminar. Se usa cuando el método necesita devolver más de un resultado, o cuando quieres separar "¿tuvo éxito?" del "¿cuál es el resultado?".

El ejemplo más famoso de .NET: `int.TryParse`. Ya conoces `int.Parse`, que truena si el texto no es un número. TryParse hace lo mismo pero sin tronar: devuelve true si pudo convertir y false si no, y el número convertido sale por el parámetro out:

```
Console.Write("Escribe tu edad: ");
string texto = Console.ReadLine();

int edad; // NO necesita inicializarse: out la va a llenar
bool pudo = int.TryParse(texto, out edad);

if (pudo)
{
    Console.WriteLine("Tienes " + edad + " años.");
}
else
{
    Console.WriteLine("Eso no es un número válido.");
}
```

¿Ves el patrón? El bool de retorno dice SI tuvo éxito; el out lleva EL RESULTADO. Es mucho más limpio que usar excepciones para controlar el flujo normal del programa. Este patrón (bool de éxito + out con el resultado) lo verás en todo .NET: `Dictionary.TryGetValue`, `DateTime.TryParse`, etc. Y fíjate: `edad` no necesitaba valor inicial, porque el contrato de out es "yo te la lleno, no me importa lo que tenía".

Ahora, cómo escribir TUS propios métodos con out:

```

static bool DividirSeguro(int a, int b, out int resultado)
{
    if (b == 0)
    {
        resultado = 0; // DEBES asignar out en TODOS los caminos
        return false; // división entre cero: falló
    }

    resultado = a / b; // camino normal: asignamos el resultado
    return true;
}

// Uso:
int cociente;
if (DividirSeguro(10, 2, out cociente))
{
    Console.WriteLine("Resultado: " + cociente); // 5
}
else
{
    Console.WriteLine("No se puede dividir entre cero.");
}

```

La regla de oro de out: el método DEBE asignar un valor al parámetro out en TODOS los caminos posibles antes de terminar. En el ejemplo, tanto la rama del if (b == 0) como el camino normal asignan `resultado`. Si olvidas asignarlo en alguna rama, el compilador no te deja compilar: "el parámetro out debe estar asignado antes de salir del método". Este es el segundo error clásico de la unidad: olvidar el out en una rama de early return. La solución: revisa cada `return` de tu método y asegúrate de que el out ya tenga valor antes de cada uno.

Diferencias ref vs out, claras de una vez:

- `ref`: la variable debe estar inicializada ANTES de llamar; el método puede leerla y modificarla. Úsalo cuando el valor inicial importa.
- `out`: la variable NO necesita estar inicializada antes; el método DEBE asignarle valor antes de terminar. Úsalo cuando el método produce un valor nuevo sin importar el anterior.
- En la llamada, ambos exigen la palabra clave (`ref` u `out`) para que sea obvio lo que pasa.

Un detalle moderno de C#: puedes declarar la variable out directamente en la llamada, sin declararla antes:


```
if (int.TryParse(texto, out int edad)) // declarada aquí mismo
{
    Console.WriteLine("Tienes " + edad + " años.");
}
```

`out int edad` declara la variable al mismo tiempo que la pasa. Más compacto, y la variable solo existe dentro del if, que es donde la necesitas (ámbito mínimo, ¿recuerdas?).

Parámetros in: solo lectura garantizada

Existe un tercer modificador, `in`, menos común pero bueno de conocer: pasa la variable por referencia (sin copiar, eficiente para estructuras grandes) pero el método NO puede modificarla: es de solo lectura. Es como decir "te presto mi receta original para que la leas, pero no la puedes rayar".

```
static double CalcularArea(in double radio)
{
    // radio = 5; // ERROR: no se puede modificar un parámetro in
    return 3.1416 * radio * radio;
}

double r = 10;
double area = CalcularArea(in r);
```

En la práctica, `in` se usa sobre todo con structs grandes por razones de rendimiento, un tema avanzado. Con saber que existe basta: parámetro in = referencia de solo lectura. Cuando lo veas en código ajeno, ya sabrás qué significa.

Sobrecarga de métodos: el mismo nombre, varias versiones

¿Te has fijado que `Console.WriteLine` acepta un int, un string, un double, o nada? No son métodos distintos con nombres distintos: es UN nombre con VARIAS versiones. Eso se llama sobrecarga (overloading): definir varios métodos con el MISMO nombre pero con DIFERENTE lista de parámetros (diferente número, o diferentes tipos, o diferente orden de tipos). C# elige automáticamente la versión correcta según los argumentos que pases en la llamada.

```

static int Sumar(int a, int b)
{
    return a + b;
}

static double Sumar(double a, double b)
{
    return a + b;
}

static int Sumar(int a, int b, int c)
{
    return a + b + c;
}

// Llamadas: C# elige la versión correcta sola
int s1 = Sumar(5, 3);           // usa la versión (int, int)
double s2 = Sumar(5.5, 3.2);    // usa la versión (double, double)
int s3 = Sumar(5, 3, 2);        // usa la versión (int, int, int)

```

La sobrecarga es como tener un ayudante que entiende varios idiomas: le hablas en español o en inglés y él responde en el que le hablaste. El que llama no necesita recordar tres nombres distintos (SumarEnteros, SumarDecimales, SumarTres): usa un solo nombre lógico y C# resuelve.

Reglas de la sobrecarga:

1. Lo que distingue una versión de otra es la FIRMA: el nombre más el número, tipos y orden de los parámetros. El tipo de RETORNO no cuenta: NO puedes tener dos métodos que solo difieran en lo que devuelven, porque en la llamada C# no sabría cuál elegir.

```

// ESTO NO COMPILA: misma firma, distinto retorno
static int Obtener() { return 1; }
static string Obtener() { return "uno"; } // ERROR

```

2. La resolución es por compatibilidad de argumentos: C# busca la versión cuyos parámetros mejor encajen con los argumentos. Si ninguna encaja, error de compilación. Si hay ambigüedad (dos versiones igual de válidas), también error: tendrás que ser más explícito.
3. Úsala cuando las versiones hacen conceptualmente LO MISMO pero con diferentes entradas. Si hacen cosas distintas, merecen nombres distintos. Sobrecargar `Procesar(int)` para sumar y `Procesar(string)` para enviar un correo sería un abuso:

mismo nombre, comportamientos sin relación. La sobrecarga debe significar "la misma operación, adaptada a diferentes datos".

Un ejemplo realista y útil: un método para mostrar un mensaje con diferentes niveles de detalle:

```
static void MostrarMensaje(string texto)
{
    Console.WriteLine(texto);
}

static void MostrarMensaje(string texto, string remitente)
{
    Console.WriteLine("[ " + remitente + " ] " + texto);
}

MostrarMensaje("Hola");           // simple
MostrarMensaje("Hola", "Sistema"); // con remitente
```

Parámetros opcionales: valores por defecto

A veces un método tiene un parámetro que casi siempre recibe el mismo valor, y solo ocasionalmente quieres cambiarlo. En lugar de sobrecargar, puedes darle un VALOR POR DEFECTO en la declaración: si el que llama no pasa ese argumento, se usa el valor por defecto automáticamente.

```
static void Saludar(string nombre, string saludo = "Hola")
{
    Console.WriteLine(saludo + ", " + nombre + ".");
}

Saludar("Ana");           // "Hola, Ana." (usa el valor por defecto)
Saludar("Luis", "Buenos días"); // "Buenos días, Luis." (lo sobreescribe)
Saludar(nombre: "Marta", saludo: "Buenas noches"); // con nombres, cualquier orden
```

Las reglas:

1. Los parámetros opcionales van AL FINAL de la lista. No puedes tener un parámetro obligatorio después de uno opcional: ¿cómo sabría C# cuál omitiste?
2. El valor por defecto debe ser una constante (un literal como "Hola", 10, true, o null). No puede ser el resultado de un cálculo.

3. Puedes combinar parámetros opcionales con argumentos nombrados para saltarte intermedios:

```
static void Configurar(string usuario, int nivel = 1, bool notificaciones = true)
{
    // ...
}

Configurar("ana"); // nivel=1, notificaciones=true
Configurar("ana", notificaciones: false); // nivel=1, notificaciones=false
```

¿Sobrecarga o parámetros opcionales? Si las versiones difieren en el NÚMERO de parámetros y las versiones con menos parámetros son solo "la completa con valores típicos", los parámetros opcionales son más simples (un solo método). Si las versiones difieren en TIPOS o hacen cosas sustancialmente distintas, usa sobrecarga. En código moderno verás ambas técnicas conviviendo sin problema.

Buenas prácticas con métodos

Cerramos la teoría con las prácticas que distinguen código profesional:

1. Métodos cortos, un solo propósito: si no puedes describir lo que hace en una frase sin usar "y", pártelo en métodos más pequeños.
2. Nombres de verbo en PascalCase: CalcularTotal, ValidarEntrada, MostrarMenu. El nombre debe decir qué hace; si necesitas leer el código para entender el nombre, el nombre está mal.
3. Pocos parámetros: más de 3 o 4 parámetros es señal de alerta. Considera agrupar datos relacionados o revisar si el método hace demasiado. (Los parámetros opcionales y los objetos ayudan aquí.)
4. Evita efectos secundarios sorpresa: un método llamado CalcularPromedio no debería ADEMÁS modificar una variable global o mostrar cosas en pantalla. Que haga lo que su nombre dice, nada más. Si necesitas que modifique algo externo, que sea explícito con ref/out, no a escondidas.
5. No repitas lógica: si el mismo bloque aparece dos veces, es un método esperando a nacer (DRY).

6. Valida en la frontera: los métodos que reciben datos del exterior (del usuario, de un archivo) deben validar sus parámetros. Los métodos internos pueden asumir datos válidos si ya fueron validados arriba. Esto evita validar lo mismo en diez lugares.
7. Prefiere devolver valores a modificar estado: un método que calcula y devuelve es más fácil de probar y reutilizar que uno que modifica variables globales por todos lados.

Ejemplos paso a paso

Ejemplo 1: Sistema de calificaciones con for, foreach y métodos

Problema: un profesor tiene las calificaciones de su grupo en un arreglo. El programa debe mostrar cada calificación con su número de alumno (usando for, porque necesita el índice), calcular el promedio (con foreach y un método), contar cuántos aprobaron (método con contador) y mostrar la calificación más alta (método con búsqueda). Todo organizado en métodos pequeños.

```
static void Main()
{
    int[] calificaciones = { 85, 92, 58, 74, 100, 67, 88, 95, 71, 60 };

    MostrarLista(calificaciones);

    double promedio = CalcularPromedio(calificaciones);
    Console.WriteLine("Promedio del grupo: " + promedio);

    int aprobados = ContarAprobados(calificaciones, 70);
    Console.WriteLine("Aprobaron: " + aprobados + " de " + calificaciones.Length);

    int maxima = BuscarMaxima(calificaciones);
    Console.WriteLine("Calificación más alta: " + maxima);
}

static void MostrarLista(int[] califs)
{
    Console.WriteLine("=== LISTA DE CALIFICACIONES ===");
    for (int i = 0; i < califs.Length; i++)
    {
        Console.WriteLine("Alumno " + (i + 1) + ": " + califs[i]);
    }
}

static double CalcularPromedio(int[] califs)
```

```

{
    int suma = 0;
    foreach (int c in califs)
    {
        suma += c;
    }
    return (double)suma / califs.Length;
}

static int ContarAprobados(int[] califs, int minimo)
{
    int contador = 0;
    foreach (int c in califs)
    {
        if (c >= minimo)
        {
            contador++;
        }
    }
    return contador;
}

static int BuscarMaxima(int[] califs)
{
    int maxima = califs[0];
    foreach (int c in califs)
    {
        if (c > maxima)
        {
            maxima = c;
        }
    }
    return maxima;
}

```

Explicación línea por línea:

- En el Main: `int[] calificaciones = { ... };` crea el arreglo con 10 calificaciones. El Main es ahora un coordinador: llama a cuatro métodos, cada uno con una responsabilidad clara. Lee el Main y entiendes TODO lo que hace el programa sin ver detalles: muestra la lista, calcula el promedio, cuenta aprobados, busca la máxima. Esa legibilidad es el premio de los métodos bien diseñados.

- `MostrarLista(calificaciones);` llama al método void pasándole el arreglo. Nota: los arreglos se pasan por referencia automáticamente (son tipos por referencia), así que el método trabaja con el arreglo original sin necesidad de ref. No lo modifica, solo lo lee.
- En `MostrarLista` usamos `for` con `i < califs.Length` porque necesitamos el ÍNDICE para mostrar "Alumno 1", "Alumno 2"... El `(i + 1)` convierte el índice base 0 a número de alumno base 1, que es lo natural para un humano. Este es el caso de libro donde `for` es obligatorio: necesitas la posición.
- `double promedio = CalcularPromedio(calificaciones);` llama al método que DEVUELVE un double y guarda el resultado. En la llamada no necesitas saber CÓMO calcula el promedio, solo que lo hace. Esa es la encapsulación: el detalle está escondido dentro del método.
- En `CalcularPromedio` usamos `foreach` porque solo necesitamos LEER cada calificación para sumarla: no nos importa la posición. `suma += c` es el acumulador clásico. El `return` convierte a double antes de dividir para no perder los decimales (¿recuerdas la división entera de la unidad 3?). Fíjate que el método no sabe ni le importa cuántos elementos hay: usa `califs.Length` y funciona con cualquier arreglo.
- `ContarAprobados(calificaciones, 70)` muestra un método con DOS parámetros: el arreglo y la calificación mínima para aprobar. Hacer el mínimo un parámetro (en lugar de escribir 70 fijo dentro del método) hace el método reutilizable: mañana puedes contar aprobados con 60 o con 80 sin tocar el método. Los valores fijos escritos dentro del código ("números mágicos") limitan la reutilización; los parámetros la expanden.
- En `ContarAprobados` vive el patrón contador: inicia en 0, se incrementa cuando se cumple la condición, se devuelve al final. `foreach` porque solo leemos.
- `BuscarMaxima` implementa el patrón de búsqueda del máximo: empieza suponiendo que el primero es el máximo (`int maxima = califs[0]`), y recorre el resto comparando: cada vez que encuentra uno mayor, actualiza. Al final, `maxima` tiene el mayor de todos. Este patrón (inicializar con el primer elemento y comparar) sirve para máximos, mínimos, y con variaciones, para muchas búsquedas. Nota: inicializar con `califs[0]` en lugar de 0 es importante: si todas las calificaciones fueran negativas (no en este caso, pero en general), inicializar en 0 daría un máximo falso de 0 que ni siquiera está en el arreglo.

Prueba mental con {85, 92, 58}: promedio = $235/3 = 78.33$; aprobados con mínimo 70: 2; máxima: 92. Todo correcto.

Ejemplo 2: Cajero automático refactorizado con métodos

Problema: el cajero de la unidad 3 funcionaba, pero todo el código estaba en el Main. Vamos a reorganizarlo en métodos: cada opción del menú es un método, y el saldo vive como campo de la clase para que todos los métodos lo compartan.

```
class Cajero
{
    static double saldo = 5000; // campo: compartido por todos los métodos

    static void Main()
    {
        int opcion;

        do
        {
            MostrarMenu();
            opcion = int.Parse(Console.ReadLine());
            Console.WriteLine();

            switch (opcion)
            {
                case 1:
                    ConsultarSaldo();
                    break;
                case 2:
                    Retirar();
                    break;
                case 3:
                    Depositar();
                    break;
                case 0:
                    Console.WriteLine("Gracias por usar el cajero.");
                    break;
                default:
                    Console.WriteLine("Opción no válida.");
                    break;
            }

            Console.WriteLine();

        } while (opcion != 0);
    }

    static void MostrarMenu()
```



```

{
    Console.WriteLine("==== CAJERO AUTOMÁTICO =====");
    Console.WriteLine("1. Consultar saldo");
    Console.WriteLine("2. Retirar");
    Console.WriteLine("3. Depositar");
    Console.WriteLine("0. Salir");
    Console.Write("Elige: ");
}

static void ConsultarSaldo()
{
    Console.WriteLine("Tu saldo es: $" + saldo);
}

static void Retirar()
{
    Console.Write("¿Cuánto quieres retirar? $");
    double monto = double.Parse(Console.ReadLine());

    if (monto <= 0)
    {
        Console.WriteLine("Error: el monto debe ser mayor que cero.");
        return; // salida temprana: no seguimos
    }

    if (monto > saldo)
    {
        Console.WriteLine("Error: saldo insuficiente.");
        return; // salida temprana
    }

    saldo -= monto;
    Console.WriteLine("Retiro exitoso. Nuevo saldo: $" + saldo);
}

static void Depositar()
{
    Console.Write("¿Cuánto quieres depositar? $");
    double monto = double.Parse(Console.ReadLine());

    if (monto <= 0)
    {
        Console.WriteLine("Error: el monto debe ser mayor que cero.");
        return;
    }
}

```

```

        saldo += monto;
        Console.WriteLine("Depósito exitoso. Nuevo saldo: $" + saldo);
    }
}

```

Explicación línea por línea:

- ``class Cajero { ... }`` declara la clase que contiene todo. ``static double saldo = 5000;`` es un CAMPO: vive fuera de los métodos y es visible para todos ellos. Es la forma correcta de compartir estado entre métodos sin andar pasando el saldo como parámetro a cada uno (que también sería válido, pero más verboso aquí).
- El Main quedó delgado: solo maneja el ciclo del menú y el switch que delega a métodos. Todo el detalle vive en los métodos.
- ``MostrarMenu()`` es un void sin parámetros: solo imprime. Si quieres cambiar el menú, lo cambias en un lugar, no en medio del ciclo.
- ``Retirar()`` usa early return dos veces: si el monto es inválido o no hay saldo, muestra el error y ``return`` termina el método ahí mismo. Solo si pasa ambas validaciones llega a ``saldo -= monto``. Compara con la versión de la unidad 3 que usaba if-else if-else anidados: esta versión es más plana y más fácil de leer.
- ``saldo -= monto`` modifica el campo directamente: como es un campo de la clase (no una variable local ni un parámetro), el cambio persiste después de que el método termina. La próxima vez que ConsultarSaldo se llame, verá el nuevo valor. Esta persistencia es exactamente para lo que sirven los campos.
- ``Depositar()`` sigue el mismo patrón: validar con early return, luego modificar. La simetría entre Retirar y Depositar hace el código predecible: quien entiende uno, entiende el otro.

Este ejemplo demuestra la refactorización: tomar código que funciona y reorganizarlo para que sea más limpio sin cambiar su comportamiento.

Ejemplo 3: Métodos con ref, out y sobrecarga - utilidades matemáticas

Problema: crear una "caja de herramientas" matemática que demuestre ref (intercambiar valores), out (división segura que devuelve dos resultados: cociente y residuo) y sobrecarga (un método Potencia con versiones para int y double).

```

static void Main()
{
    // 1. ref: intercambiar dos valores
    int a = 5, b = 10;
}

```

```

Console.WriteLine("Antes: a=" + a + ", b=" + b);
Intercambiar(ref a, ref b);
Console.WriteLine("Después: a=" + a + ", b=" + b);

// 2. out: división que devuelve cociente Y residuo
int cociente, residuo;
if (Dividir(17, 5, out cociente, out residuo))
{
    Console.WriteLine("17 / 5 = " + cociente + " con residuo " + residuo);
}

if (!Dividir(10, 0, out cociente, out residuo))
{
    Console.WriteLine("No se puede dividir entre cero.");
}

// 3. Sobrecarga: Potencia elige la versión según los tipos
Console.WriteLine("2^10 = " + Potencia(2, 10));    // versión int
Console.WriteLine("2.5^3 = " + Potencia(2.5, 3));  // versión double
}

static void Intercambiar(ref int x, ref int y)
{
    int temp = x; // guardamos x en una variable temporal
    x = y;        // x toma el valor de y
    y = temp;     // y toma el valor guardado de x
}

static bool Dividir(int dividendo, int divisor, out int cociente, out int residuo)
{
    if (divisor == 0)
    {
        cociente = 0; // out debe asignarse en TODOS los caminos
        residuo = 0;
        return false;
    }

    cociente = dividendo / divisor;
    residuo = dividendo % divisor;
    return true;
}

static int Potencia(int base_, int exponente)
{
    int resultado = 1;

```

```

    for (int i = 0; i < exponente; i++)
    {
        resultado *= base_;
    }
    return resultado;
}

static double Potencia(double base_, int exponente)
{
    double resultado = 1.0;
    for (int i = 0; i < exponente; i++)
    {
        resultado *= base_;
    }
    return resultado;
}

```

Explicación línea por línea:

- ``Intercambiar(ref a, ref b)``: sin ref esto sería imposible, porque los parámetros serían copias y el intercambio se perdería al terminar el método. Con ref, x y y SON a y b: el intercambio es real. El algoritmo clásico usa una variable temporal: guardas x en temp, pasas y a x, pasas temp a y. Sin la temporal perderías un valor. Este truco aparece en algoritmos de ordenamiento: memorízalo.
- Fíjate que en la llamada escribimos ``ref`` dos veces (uno por cada parámetro), y que ``a`` y ``b`` estaban inicializadas (5 y 10): requisito de ref.
- ``Dividir`` demuestra por qué out existe: un método solo puede tener UN valor de retorno, pero aquí necesitamos DOS resultados (cociente y residuo). Con dos parámetros out, el método "devuelve" tres cosas: el bool (¿tuvo éxito?) más los dos outs. El operador ``%`` (módulo) da el residuo: $17 \% 5 = 2$.
- En la rama de error (divisor == 0) asignamos AMBOS out antes del return false. Si olvidaras uno, no compila: cada return de un método con out debe tener todos los out asignados antes.
- En el Main, ``int cociente, residuo;`` las declara SIN inicializar: con out está permitido, porque el método las va a llenar. Luego ``Dividir(17, 5, out cociente, out residuo)``: la palabra out en la llamada avisa "estas variables van a ser llenadas por el método".
- La sobrecarga de Potencia: dos métodos con el mismo nombre, distinta firma. ``Potencia(2, 10)`` pasa dos int: C# elige la versión int. ``Potencia(2.5, 3)`` pasa un double y un int: elige la versión double. El que llama no piensa en versiones: solo llama a Potencia y funciona. Nota que la lógica es idéntica en ambas versiones; solo cambian los tipos. En

código real, a veces la versión double delega a la int o viceversa para no duplicar lógica, pero para el ejemplo la duplicación pequeña es aceptable y clara.

- El algoritmo de potencia: multiplica la base por sí misma `exponente` veces. `resultado` empieza en 1 (acumulador multiplicativo: ¡siempre inicia en 1, no en 0!). Con exponente 0, el for da cero vueltas y devuelve 1: correcto, porque cualquier número a la potencia 0 es 1. El acumulador bien inicializado maneja el caso borde gratis.

Ejemplo 4: Generador de reportes con parámetros opcionales y argumentos nombrados

Problema: un método que genera un reporte de ventas con opciones configurables (título, si muestra detalles, moneda). La mayoría de las veces se usa con valores típicos, así que los parámetros opcionales simplifican las llamadas comunes.

```
static void Main()
{
    double[] ventas = { 1500.50, 2300.75, 1800.00, 3100.25 };

    GenerarReporte(ventas); // todo por defecto
    Console.WriteLine();

    GenerarReporte(ventas, "REPORTE MENSUAL"); // título personalizado
    Console.WriteLine();

    GenerarReporte(ventas, mostrarDetalles: true, moneda: "USD"); // con nombres
}

static void GenerarReporte(double[] datos,
                           string titulo = "REPORTE DE VENTAS",
                           bool mostrarDetalles = false,
                           string moneda = "MXN")
{
    Console.WriteLine("=== " + titulo + " ===");

    if (mostrarDetalles)
    {
        for (int i = 0; i < datos.Length; i++)
        {
            Console.WriteLine("Venta " + (i + 1) + ": " + moneda + " " + datos[i]);
        }
    }

    double total = 0;
```

```

double maxima = datos[0];

foreach (double v in datos)
{
    total += v;
    if (v > maxima)
    {
        maxima = v;
    }
}

double promedio = total / datos.Length;

Console.WriteLine("Total: " + moneda + " " + total);
Console.WriteLine("Promedio: " + moneda + " " + promedio);
Console.WriteLine("Venta máxima: " + moneda + " " + maxima);
Console.WriteLine("Número de ventas: " + datos.Length);
}

```

Explicación línea por línea:

- La declaración tiene UN parámetro obligatorio (`datos`) y TRES opcionales con valores por defecto. Los opcionales van al final, como exige la regla.
- `GenerarReporte(ventas);` usa todos los valores por defecto: título "REPORTE DE VENTAS", sin detalles, moneda MXN. La llamada más simple posible para el caso más común.
- `GenerarReporte(ventas, "REPORTE MENSUAL");` personaliza solo el título; lo demás sigue por defecto. Los parámetros opcionales se llenan de izquierda a derecha: no puedes saltarte el título para personalizar solo la moneda... a menos que uses argumentos nombrados.
- `GenerarReporte(ventas, mostrarDetalles: true, moneda: "USD");` usa argumentos nombrados para saltarse el título (que queda por defecto) y personalizar los otros dos. Esta combinación (opcionales + nombrados) es extremadamente expresiva: la llamada dice exactamente qué está configurando.
- Dentro del método: si `mostrarDetalles` es true, un for con índice muestra cada venta numerada (necesitamos el índice para "Venta 1", "Venta 2"). Si es false, se salta esa sección.
- El foreach calcula total y máxima en UNA sola pasada: dos acumuladores en el mismo ciclo. Esto es eficiente y es un patrón común: varios cálculos agregados en un solo recorrido.

- Nota de precisión: C# distingue mayúsculas de minúsculas (es case-sensitive). ``Console.WriteLine`` lleva W mayúscula; si escribes ``writeLine`` con minúscula, no compila. Este es el tipo de error que a las 2 de la mañana te toma 10 minutos encontrar si no sabes que el lenguaje es sensible a mayúsculas. Cuando el compilador diga que algo "no existe", revisa primero las mayúsculas.

Ejemplo 5: Búsqueda en arreglo con early return y break

Problema: buscar un nombre en una lista de invitados. Si lo encuentras, muestra su posición y termina; si no está, avísalo. El método devuelve la posición o -1 si no existe (el -1 como "no encontrado" es una convención clásica en programación).

```

static void Main()
{
    string[] invitados = { "Ana", "Luis", "Marta", "Pedro", "Sofía" };

    int pos = BuscarInvitado(invitados, "Marta");
    if (pos != -1)
    {
        Console.WriteLine("Marta está en la posición " + pos + ".");
    }
    else
    {
        Console.WriteLine("Marta no está en la lista.");
    }

    pos = BuscarInvitado(invitados, "Juan");
    if (pos != -1)
    {
        Console.WriteLine("Juan está en la posición " + pos + ".");
    }
    else
    {
        Console.WriteLine("Juan no está en la lista.");
    }
}

static int BuscarInvitado(string[] lista, string nombre)
{
    for (int i = 0; i < lista.Length; i++)
    {
        if (lista[i] == nombre)
        {
            return i; // ¡encontrado! devolvemos la posición y terminamos
        }
    }

    return -1; // terminamos el ciclo sin encontrar: no existe
}

```

Explicación línea por línea:

- `BuscarInvitado` encapsula la búsqueda lineal: recorre el arreglo comparando cada elemento. En cuanto encuentra coincidencia, `return i` devuelve la posición Y termina el método de inmediato: no necesita break porque el return ya sale del método completo. Es el early return aplicado a búsquedas: limpio y eficiente.

- Si el for termina sin haber retornado, significa que ningún elemento coincidió: ``return -1`` señala "no encontrado". El -1 es seguro como señal porque ningún índice válido es negativo.
- En el Main, el patrón de uso es siempre el mismo: llamar, comparar contra -1, actuar. Este método es reutilizable para buscar cualquier nombre en cualquier arreglo de strings.
- Fíjate en la comparación ``lista[i] == nombre`` con strings: compara el CONTENIDO del texto, no la referencia. En C#, ``==`` con strings hace lo correcto (compara carácter por carácter). Es sensible a mayúsculas: "marta" no coincidiría con "Marta". Si quieres búsqueda insensible a mayúsculas, compara ``lista[i].ToLower() == nombre.ToLower()``.
- Este ejemplo también muestra algo sutil: el método NO necesita saber el tamaño del arreglo. Usa `lista.Length` y funciona con 5 invitados o con 5000. Los métodos que operan sobre arreglos usando Length en lugar de números fijos son reutilizables por diseño.

Práctica

Los ejercicios van de menor a mayor dificultad. Resuelve cada uno por tu cuenta antes de mirar la solución: si te atorras más de 20 minutos, estudia la solución línea por línea, ciérrala y reescríbela tú solo.

Ejercicios

1. Contador de vocales: pide una palabra al usuario y cuenta cuántas vocales tiene usando un foreach que recorra cada carácter del string. Considera mayúsculas y minúsculas (convierte cada letra a minúscula antes de comparar). Muestra el total. Pista: un string se puede recorrer con foreach como si fuera una colección de char.
2. Fibonacci con for: la serie de Fibonacci empieza con 0 y 1, y cada número siguiente es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13... Pide un número n y muestra los primeros n términos usando un for. Organiza el cálculo en un método ``MostrarFibonacci(int n)` void`.
3. Calculadora con métodos: escribe una calculadora que pida dos números y una operación (+, -, *, /). Cada operación debe ser un método separado: ``Sumar``, ``Restar``, ``Multiplicar``, ``Dividir``. El método Dividir debe manejar la división entre cero devolviendo false y usando un parámetro out para el resultado (patrón TryParse). Usa switch para elegir la operación.
4. Promedio con validación: modifica el método CalcularPromedio del Ejemplo 1 para que ignore las calificaciones fuera del rango 0-100 (use continue para saltarlas) y devuelva el promedio solo de las válidas. Si no hay ninguna válida, debe indicarlo: usa un parámetro

out bool para señalar si el cálculo fue posible, o devuelve -1 como señal. Decide cuál diseño te parece mejor y justifica tu elección en un comentario.

5. Intercambio sin ref (piénsalo): escribe un método ``void IntentarIntercambiar(int a, int b)`` SIN ref que intente intercambiar dos valores, llámalo y observa que NO funciona (los originales no cambian). Luego escribe la versión con ref que SÍ funciona. En un comentario explica con tus palabras por qué la primera versión falla. Este ejercicio es para que el paso por valor te quede grabado a fuego.
6. Sobrecarga de áreas: crea tres versiones sobrecargadas del método ``CalcularArea``: una para cuadrado (recibe el lado), una para rectángulo (recibe base y altura) y una para círculo (recibe el radio y un parámetro bool extra ``esCirculo`` para distinguir la firma, o mejor: piensa otra forma de distinguir las). Muestra un menú que pida la figura y los datos, y llame a la versión correcta. Reflexiona: ¿qué firma le pondrías a cada una para que no haya ambigüedad?
7. Lista de compras con foreach (reto): crea una lista de productos usando dos arreglos paralelos: ``string[] productos`` y ``double[] precios`` (mismo índice = mismo producto). Escribe métodos: ``MostrarTicket`` (usa for con índice para mostrar número, producto y precio), ``CalcularTotal`` (usa foreach para sumar) y ``BuscarMasCaro`` (devuelve el ÍNDICE del producto más caro usando for, porque necesitas la posición; luego el Main usa ese índice para mostrar el nombre). Este ejercicio practica cuándo for y cuándo foreach.
8. Adivinanzas con intentos configurables (reto integrador): combina todo: un método ``JugarAdivinanza(int minimo = 1, int maximo = 50, int intentos = 6)`` con parámetros opcionales que implemente el juego de adivinar el número. Debe devolver true si el jugador ganó y false si perdió (para que el Main muestre el mensaje final). Usa Random para el secreto, while para los intentos, switch expression para las pistas de temperatura (como en la unidad 3) y un contador de "cerca". El Main llama al método tres veces: una con valores por defecto, una solo cambiando el rango a 1-100, y una con argumentos nombrados para 8 intentos.

Soluciones

Solución 1: Contador de vocales

```

Console.Write("Escribe una palabra: ");
string palabra = Console.ReadLine();

int vocales = 0;

foreach (char letra in palabra)
{
    char minuscula = char.ToLower(letra);

    if (minuscula == 'a' || minuscula == 'e' || minuscula == 'i' ||
        minuscula == 'o' || minuscula == 'u')
    {
        vocales++;
    }
}

Console.WriteLine("La palabra tiene " + vocales + " vocales.");

```

Explicación: el foreach recorre el string carácter por carácter (un string es una colección de char). `char.ToLower` convierte cada letra a minúscula para comparar una sola vez en lugar de comparar contra 'A' y 'a' por separado. La condición con `||` verifica las 5 vocales. Las comillas SIMPLES ('a') son para char; las DOBLES ("a") son para string: no las confundas. El contador `vocales` se declara antes del ciclo y se incrementa dentro: el patrón contador clásico aplicado a un foreach.

Solución 2: Fibonacci con for

```

static void Main()
{
    Console.Write("¿Cuántos términos de Fibonacci quieres ver? ");
    int n = int.Parse(Console.ReadLine());
    MostrarFibonacci(n);
}

static void MostrarFibonacci(int n)
{
    if (n <= 0)
    {
        Console.WriteLine("El número debe ser mayor que cero.");
        return; // salida temprana: nada que mostrar
    }

    int anterior = 0; // primer término
    int actual = 1;   // segundo término

    Console.Write("Serie: ");

    for (int i = 0; i < n; i++)
    {
        Console.Write(anterior + " ");

        int siguiente = anterior + actual; // el siguiente es la suma
        anterior = actual;                 // avanzamos: el actual pasa a ser
anterior                                // y el siguiente pasa a ser actual
        actual = siguiente;
    }

    Console.WriteLine();
}

```

Explicación: el método es void porque su trabajo es MOSTRAR, no calcular un valor para devolver. La validación con early return protege contra $n \leq 0$. El algoritmo mantiene dos variables (anterior y actual) y en cada vuelta: muestra anterior, calcula el siguiente como la suma, y "avanza" las variables. Es un patrón de ventana deslizante: solo necesitas recordar los últimos dos valores, no toda la serie. Prueba mental con $n=5$: anterior=0, actual=1. Vuelta 1: muestra 0, siguiente=1, anterior=1, actual=1. Vuelta 2: muestra 1, siguiente=2, anterior=1, actual=2. Vuelta 3: muestra 1, siguiente=3, anterior=2, actual=3. Vuelta 4: muestra 2, siguiente=5, anterior=3, actual=5. Vuelta 5: muestra 3... Serie: 0 1 1 2 3. Correcto.

Solución 3: Calculadora con métodos

```
static void Main()
{
    Console.Write("Primer número: ");
    double a = double.Parse(Console.ReadLine());

    Console.Write("Segundo número: ");
    double b = double.Parse(Console.ReadLine());

    Console.Write("Operación (+, -, *, /): ");
    string op = Console.ReadLine();

    switch (op)
    {
        case "+":
            Console.WriteLine("Resultado: " + Sumar(a, b));
            break;
        case "-":
            Console.WriteLine("Resultado: " + Restar(a, b));
            break;
        case "*":
            Console.WriteLine("Resultado: " + Multiplicar(a, b));
            break;
        case "/":
            double resultado;
            if (Dividir(a, b, out resultado))
            {
                Console.WriteLine("Resultado: " + resultado);
            }
            else
            {
                Console.WriteLine("Error: no se puede dividir entre cero.");
            }
            break;
        default:
            Console.WriteLine("Operación no válida.");
            break;
    }
}

static double Sumar(double a, double b)
{
    return a + b;
}
```

```

static double Restar(double a, double b)
{
    return a - b;
}

static double Multiplicar(double a, double b)
{
    return a * b;
}

static bool Dividir(double a, double b, out double resultado)
{
    if (b == 0)
    {
        resultado = 0; // out asignado en el camino de error
        return false;
    }

    resultado = a / b;
    return true;
}

```

Explicación: cada operación es un método con un solo propósito y nombre de verbo. Sumar, Restar y Multiplicar devuelven directamente: son funciones puras (mismos argumentos, mismo resultado, sin efectos secundarios). Dividir sigue el patrón TryParse: el bool indica éxito y el out lleva el resultado, separando el "¿se pudo?" del "¿cuánto dio?". El switch dirige a cada método y su default atrapa operaciones inválidas. Si mañana quieres agregar potencia o módulo, agregas un método y un case: el Main apenas cambia. Esa extensibilidad es el fruto de organizar en métodos.

Solución 4: Promedio con validación

```

// Diseño elegido: out bool para indicar éxito, porque -1 como señal
// es ambiguo (¿y si el promedio real pudiera ser -1 en otro contexto?).
// El out deja el valor de retorno limpio para el promedio real.

static double CalcularPromedioValidado(int[] califs, out bool exito)
{
    int suma = 0;
    int validas = 0;

    foreach (int c in califs)
    {
        if (c < 0 || c > 100)
        {
            continue; // calificación inválida: se salta, no se cuenta
        }

        suma += c;
        validas++;
    }

    if (validas == 0)
    {
        exito = false;
        return 0; // valor sin significado; el que llama debe revisar exito
    }

    exito = true;
    return (double)suma / validas;
}

// Uso:
int[] datos = { 85, 150, 92, -5, 78 };
bool ok;
double prom = CalcularPromedioValidado(datos, out ok);

if (ok)
{
    Console.WriteLine("Promedio de válidas: " + prom); // (85+92+78)/3 = 85
}
else
{
    Console.WriteLine("No había calificaciones válidas.");
}

```

Explicación: el `continue` filtra las inválidas sin anidar el resto del código: si la calificación está fuera de rango, se salta la vuelta y pasa a la siguiente. Solo las válidas llegan al acumulador y al contador. El caso borde (ninguna válida) se maneja con early return que asigna el out antes de salir. La justificación del diseño en el comentario es importante: en código profesional, estas decisiones se documentan porque el "por qué" se olvida más rápido que el "qué". Nota que el que llama SIEMPRE debe revisar `ok` antes de usar `prom`: un contrato claro entre el método y quien lo usa.

Solución 5: Intercambio sin ref (piénsalo)

```
static void Main()
{
    int x = 5, y = 10;

    IntentarIntercambiar(x, y);
    Console.WriteLine("Sin ref: x=" + x + ", y=" + y); // x=5, y=10: NO cambió

    IntercambiarDeVerdad(ref x, ref y);
    Console.WriteLine("Con ref: x=" + x + ", y=" + y); // x=10, y=5: Sí cambió
}

static void IntentarIntercambiar(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    // El intercambio SÍ ocurre aquí adentro, pero a y b son COPIAS.
    // Cuando el método termina, las copias se destruyen y las
    // originales (x, y) nunca se enteraron. Por eso no funciona:
    // el paso por valor protege las variables del que llama.
}

static void IntercambiarDeVerdad(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
    // Con ref, a y b SON las variables originales (x, y).
    // No hay copias: el intercambio modifica directamente
    // la memoria del que llamó. Por eso sí funciona.
}
```


Explicación: el código de ambos métodos es IDÉNTICO; lo único que cambia es el `ref`. Esa es la lección: el algoritmo estaba bien en ambos casos, la diferencia es puramente el mecanismo de paso de parámetros. La versión sin ref intercambia copias que mueren al terminar el método. Este experimento vale más que mil explicaciones teóricas: cuando LO VES fallar y luego LO VES funcionar con el único cambio del ref, el concepto queda grabado para siempre.

Solución 6: Sobrecarga de áreas

```
static void Main()
{
    Console.WriteLine("1. Cuadrado  2. Rectángulo  3. Círculo");
    Console.Write("Elige figura: ");
    int fig = int.Parse(Console.ReadLine());

    switch (fig)
    {
        case 1:
            Console.Write("Lado: ");
            double lado = double.Parse(Console.ReadLine());
            Console.WriteLine("Área: " + CalcularArea(lado));
            break;
        case 2:
            Console.Write("Base: ");
            double b = double.Parse(Console.ReadLine());
            Console.Write("Altura: ");
            double h = double.Parse(Console.ReadLine());
            Console.WriteLine("Área: " + CalcularArea(b, h));
            break;
        case 3:
            Console.Write("Radio: ");
            double r = double.Parse(Console.ReadLine());
            Console.WriteLine("Área: " + CalcularArea(r, true));
            break;
        default:
            Console.WriteLine("Figura no válida.");
            break;
    }
}

static double CalcularArea(double lado) // cuadrado: un parámetro
{
    return lado * lado;
}
```

```

static double CalcularArea(double base_, double altura) // rectángulo: dos parámetros
{
    return base_ * altura;
}

static double CalcularArea(double radio, bool esCirculo) // círculo: dos parámetros,
tipos distintos
{
    return 3.1416 * radio * radio;
}

```

Explicación: las tres firmas son distintas: (double), (double, double) y (double, bool). C# distingue la del rectángulo de la del círculo por el TIPO del segundo parámetro, no solo por la cantidad. El parámetro `esCirculo` no se usa dentro del método: existe solo para hacer la firma única. Como ejercicio de sobrecarga es válido y demuestra que la firma incluye los TIPOS. La reflexión pedida: si dos versiones tuvieran la misma firma, habría ambigüedad y no compilaría; el tipo de retorno no sirve para distinguirlas.

Solución 7: Lista de compras con foreach (reto)

```

static void Main()
{
    string[] productos = { "Leche", "Pan", "Huevos", "Queso", "Manzanas" };
    double[] precios = { 25.50, 18.00, 45.00, 89.90, 32.75 };

    MostrarTicket(productos, precios);

    double total = CalcularTotal(precios);
    Console.WriteLine("TOTAL: $" + total);

    int indiceCaro = BuscarMasCaro(precios);
    Console.WriteLine("Lo más caro: " + productos[indiceCaro] +
        " ($" + precios[indiceCaro] + ")");
}

static void MostrarTicket(string[] productos, double[] precios)
{
    Console.WriteLine("=== TICKET DE COMPRA ===");
    for (int i = 0; i < productos.Length; i++) // for: necesita el índice
    {
        Console.WriteLine((i + 1) + ". " + productos[i] + " ... $" + precios[i]);
    }
}

```

```

static double CalcularTotal(double[] precios)
{
    double total = 0;
    foreach (double p in precios) // foreach: solo suma, no necesita índice
    {
        total += p;
    }
    return total;
}

static int BuscarMasCaro(double[] precios)
{
    int indiceMax = 0;
    for (int i = 1; i < precios.Length; i++) // for: necesita la POSICIÓN del máximo
    {
        if (precios[i] > precios[indiceMax])
        {
            indiceMax = i;
        }
    }
    return indiceMax;
}

```

Explicación: este ejercicio es un examen disfrazado de la regla for-vs-foreach. MostrarTicket usa for porque necesita el índice para numerar y para emparejar productos[i] con precios[i]: los arreglos paralelos SOLO funcionan con índices. CalcularTotal usa foreach porque solo suma. BuscarMasCaro usa for porque debe devolver la POSICIÓN del máximo: el Main necesita el índice para buscar el nombre en el otro arreglo. Fíjate que el ciclo empieza en i = 1: como indiceMax ya es 0, comparar precios[0] consigo mismo sería inútil.

Solución 8: Adivinanzas con intentos configurables (reto integrador)

```

static void Main()
{
    Console.WriteLine("--- Juego 1: clásico ---");
    bool gano1 = JugarAdivinanza();
    Console.WriteLine(gano1 ? "¡Ganaste!" : "Perdiste. Suerte para la próxima.");

    Console.WriteLine("--- Juego 2: del 1 al 100 ---");
    bool gano2 = JugarAdivinanza(1, 100);
    Console.WriteLine(gano2 ? "¡Ganaste!" : "Perdiste. Suerte para la próxima.");

    Console.WriteLine("--- Juego 3: modo difícil, 8 intentos ---");
}

```

```

    bool gano3 = JugarAdivinanza(minimo: 1, maximo: 100, intentos: 8);
    Console.WriteLine(gano3 ? "¡Ganaste!" : "Perdiste. Suerte para la próxima.");
}

static bool JugarAdivinanza(int minimo = 1, int maximo = 50, int intentos = 6)
{
    Random azar = new Random();
    int secreto = azar.Next(minimo, maximo + 1);
    int intento = 1;
    int vecesCerca = 0;

    Console.WriteLine("Adivina el número (" + minimo + "-" + maximo + "). " +
        "Tienes " + intentos + " intentos.");

    while (intento <= intentos)
    {
        Console.Write("Intento " + intento + ": ");
        int propuesta = int.Parse(Console.ReadLine());

        if (propuesta == secreto)
        {
            Console.WriteLine("¡Exacto! Lo lograste en " + intento + " intentos.");
            Console.WriteLine("Estuviste cerca " + vecesCerca + " veces.");
            return true; // ganó: salimos con true
        }

        int diferencia = Math.Abs(propuesta - secreto);

        string pista = diferencia switch
        {
            <= 3 => "Hirviendo",
            <= 8 => "Caliente",
            <= 15 => "Tibio",
            _ => "Frío"
        };
        Console.WriteLine(pista + "...");

        if (diferencia <= 8)
        {
            vecesCerca++;
        }

        Console.WriteLine(propuesta < secreto ? "Es MAYOR." : "Es MENOR.");
        intento++;
    }
}

```

```
Console.WriteLine("Se acabaron los intentos. Era el " + secreto + ".");
Console.WriteLine("Estuviste cerca " + vecesCerca + " veces.");
return false; // perdió: salimos con false
}
```

Explicación: el método integra TODA la unidad: parámetros opcionales, while con contador de intentos, switch expression con patrones relacionales, contador de cercanía, operador ternario para la pista de mayor/menor, y early return con `return true` en cuanto adivina (no necesita bandera). El bool de retorno le dice al Main el resultado para que ÉL decida el mensaje final: el método calcula, el que llama presenta. Las tres llamadas muestran la flexibilidad: `JugarAdivinanza()` usa todo por defecto; `JugarAdivinanza(1, 100)` cambia los dos primeros por posición; la tercera usa argumentos nombrados. Un solo método, tres juegos distintos, cero duplicación.

Resumen de la unidad

- El `for` (inicialización; condición; actualización) junta en un lugar las tres piezas del ciclo contado; el patrón estándar `for (int i = 0; i < n; i++)` da exactamente n vueltas y es el ciclo más escrito en programación.
- Usa `for` cuando sabes cuántas vueltas darás antes de empezar; usa `while` cuando la salida depende de algo que ocurre dentro del ciclo; usa `do-while` cuando el bloque debe ejecutarse al menos una vez.
- Cuidado con el error por uno (off-by-one): no mezcles inicios en 0 con condiciones `<=`; elige un patrón consistente y verifícalo con un caso pequeño.
- Los arreglos guardan varios valores del mismo tipo numerados desde 0; `Length` dice cuántos elementos hay y el último índice válido es `Length - 1`; salirte del rango produce `IndexOutOfRangeException`.
- El `foreach` (tipo variable in colección) recorre cada elemento sin índices: es más simple y seguro cuando solo necesitas leer; la variable de recorrido es de solo lectura.
- Usa `foreach` para leer elementos sin importar la posición; usa `for` cuando necesites el índice, modificar elementos, recorrer hacia atrás o trabajar con arreglos paralelos.
- Nunca agregues ni quites elementos de una colección dentro de un `foreach`: el programa truena con `InvalidOperationException`; para quitar, recorre con `for` de atrás hacia adelante.
- `break` abandona el ciclo completo (ideal en búsquedas); `continue` salta a la siguiente vuelta (ideal para filtrar al inicio del bloque); no abusos de ellos.

- Un método se DECLARA una vez (qué hace) y se LLAMA cuantas veces quieras (úsalo); la declaración incluye modificador, tipo de retorno, nombre, parámetros y cuerpo.
- Los métodos se nombran con verbos en PascalCase (CalcularTotal, MostrarMenu); los métodos void hacen acciones sin devolver nada; el ``return`` entrega el resultado y termina el método.
- Por defecto los parámetros son copias (paso por valor): el método no puede modificar tus variables; la distinción parámetro (declaración) vs argumento (llamada) te hace preciso.
- El ámbito de una variable es donde existe: declara cada variable en el ámbito más pequeño posible; los campos de clase comparten estado entre métodos.
- Métodos cortos con un solo propósito: si su descripción necesita la palabra "y", pártelo; no repitas lógica (DRY); válida en la frontera.
- ``ref`` pasa la variable original (debe estar inicializada; se escribe en declaración y llamada); úsalo cuando el método lee Y modifica.
- ``out`` obliga al método a asignar el valor en todos los caminos (la variable no necesita valor inicial); úsalo cuando el método produce un valor nuevo, como en el patrón TryParse (bool de éxito + out con resultado).
- ``in`` pasa referencia de solo lectura: el método no puede modificar el parámetro; útil para rendimiento con datos grandes.
- La sobrecarga define varias versiones del mismo método con distinta firma (número, tipos u orden de parámetros); el tipo de retorno no distingue versiones; úsala cuando las versiones hacen lo mismo con diferentes entradas.
- Los parámetros opcionales llevan valores por defecto y van al final; combinados con argumentos nombrados hacen llamadas flexibles y legibles.

UNIDAD 5: Arreglos y POO básica

Imagina que tienes que escribir el programa de un supermercado. El supermercado tiene diez mil productos, cada uno con su precio. ¿Cómo guardarías diez mil precios en tu programa? La primera idea que se le ocurre a un principiante es crear diez mil variables: precio1, precio2, precio3... hasta precio10000. Esa idea funciona en teoría, pero en la práctica es un desastre: escribirías diez mil líneas, tu código sería imposible de mantener y si mañana el supermercado agrega cien productos más, tendrías que editar el programa a mano. Claramente necesitas algo mejor: necesitas una forma de guardar muchos valores del mismo tipo bajo un solo nombre, de manera ordenada, y de recorrerlos todos sin repetir código. Eso es exactamente lo que hacen los arreglos.

En esta unidad vas a dar un salto grande en tu formación como programador. Primero dominarás los arreglos: unidimensionales y multidimensionales, cómo usar índices, la propiedad Length, cómo recorrerlos con ciclos, y qué son las matrices. Con los arreglos dejarás atrás el hábito de crear una variable por cada dato y empezarás a pensar como programador profesional: estructuras que crecen, que se recorren solas y que representan colecciones reales de datos. Después, en la segunda mitad de la unidad, darás el salto más importante de todo este curso: la programación orientada a objetos, o POO.

La POO es una forma de organizar tu código pensando en "cosas" del mundo real en lugar de pensar solo en instrucciones sueltas. Hasta ahora tus programas eran listas de pasos: lee un número, calcula algo, imprime un resultado. Eso está bien para programas pequeños, pero cuando un programa crece a miles o millones de líneas, una lista de pasos se vuelve inmanejable. La POO resuelve esto agrupando datos y comportamientos en unidades llamadas clases, y a partir de esas clases creas objetos, que son ejemplares concretos. Un objeto Estudiante tiene un nombre, una matrícula y puede inscribirse a un curso. Un objeto Producto tiene un precio, un stock y puede aplicarse un descuento. Así se modela el mundo real dentro del código.

Aprenderás qué es una clase y qué es un objeto y por qué no son lo mismo (una de las confusiones más comunes entre principiantes), cómo declarar campos y métodos de instancia, cómo usar la palabra `new` para crear objetos, y la analogía clásica del plano de casa que te va a acompañar toda tu carrera. Después conocerás los constructores: esos métodos especiales que se ejecutan al nacer un objeto, con o sin parámetros, el constructor por defecto que C# te regala cuando no escribes ninguno, y la palabra `this` para referirte al objeto actual. También verás los errores más típicos: el temido índice fuera de rango (`IndexOutOfRangeException`), la `NullReferenceException` que aparece cuando intentas usar un objeto que nunca nació, y la confusión entre clase y objeto. Y por supuesto, las buenas prácticas que usan los profesionales.

Al terminar esta unidad podrás hacer cosas que antes eran imposibles o muy engorrosas para ti: manejar listas de datos como calificaciones, precios o nombres con arreglos; recorrer miles de datos con unos cuantos renglones de código; representar entidades del mundo real con clases y objetos bien diseñados; crear objetos con valores iniciales usando constructores; y leer y entender código orientado a objetos, que es como está escrito casi todo el software profesional que usas a diario: aplicaciones de escritorio, videojuegos, apps de banco, sistemas de tiendas. Esta unidad es la puerta de entrada a la programación de verdad. Tómate tu tiempo con ella.

Explicación detallada

Clase 13: Arreglos unidimensionales - tu primera colección de datos

Un arreglo es una estructura de datos que guarda varios valores del mismo tipo bajo un solo nombre, uno tras otro, en posiciones numeradas llamadas índices. Piensa en un arreglo como un librero: el librero es uno solo (el nombre del arreglo), pero tiene varios estantes numerados (los índices), y en cada estante guardas un libro (cada valor). Para llegar al libro del estante número 3 no necesitas recorrer todo el librero: vas directo al estante 3. Así de directo es el acceso a un arreglo: si quieres el tercer elemento, lo pides por su índice y listo, sin importar si el arreglo tiene diez o diez millones de elementos. Esa velocidad de acceso directo es una de las razones por las que los arreglos son tan fundamentales.

En C#, un arreglo se declara con corchetes después del tipo. Si quieres un arreglo de enteros con capacidad para 5 elementos, escribes: `int[] calificaciones = new int[5];`. Analicemos esta línea con calma porque tiene varias partes importantes. La parte `int[]` dice "esto es un arreglo de enteros". El nombre `calificaciones` es el nombre del librero. La parte `new int[5]` crea realmente el espacio en memoria para 5 enteros y, ojo, esto es importante: `new` aquí no es

opcional. Los arreglos en C# son objetos, y como todo objeto necesitan ser creados con `new` antes de usarse. Si declaras `int[] calificaciones`; sin el `new`, solo tienes una variable vacía que apunta a nada, y si intentas usarla obtendrás una `NullReferenceException`, de la que hablaremos más adelante en esta unidad.

Ahora viene el detalle que más confunde a los principiantes de todo el mundo: en C# (y en casi todos los lenguajes) los índices empiezan en cero, no en uno. El primer elemento está en la posición 0, el segundo en la posición 1, y así sucesivamente. Entonces en un arreglo de 5 elementos, los índices válidos son 0, 1, 2, 3 y 4. No existe el índice 5 en un arreglo de 5 elementos. Esto se llama indexación base cero y tiene razones históricas y técnicas, pero para ti lo importante es grabarlo a fuego: el primer elemento es el cero, el último elemento es el tamaño menos uno. Cuando recién empiezas, vas a cometer el error de usar el índice 5 en un arreglo de 5 elementos. Todos lo hemos hecho. Con la práctica se vuelve natural.

Para guardar un valor en una posición usas el operador de corchetes: `calificaciones[0] = 85`; guarda el 85 en la primera posición. Para leerlo, usas la misma sintaxis: `Console.WriteLine(calificaciones[0])`; imprime 85. También puedes crear un arreglo ya con valores iniciales, sin escribir el tamaño: `int[] numeros = { 10, 20, 30, 40, 50 };`. Aquí C# cuenta los valores y crea el arreglo del tamaño exacto. Esta forma es muy cómoda cuando ya conoces los datos de antemano.

Una pregunta natural es: ¿qué valores tiene un arreglo recién creado con `new`? C# inicializa automáticamente cada posición con el valor por defecto del tipo: 0 para los numéricos, `false` para `bool`, y `null` para los tipos de referencia como `string`. Así que `new int[5]` te da cinco ceros. Esto es útil pero también es una trampa: si lees una posición que nunca llenaste, obtienes un cero silencioso en lugar de un error, y ese cero puede colarse en tus cálculos sin que te des cuenta. Por eso es buena práctica llenar el arreglo antes de usarlo, o al menos saber qué posiciones ya tienen datos reales.

La propiedad `Length` te dice cuántos elementos tiene el arreglo: `calificaciones.Length` devuelve 5. `Length` es tu mejor amiga cuando recorres arreglos, porque te evita tener que recordar el tamaño a mano. Un ciclo `for` clásico para recorrer un arreglo se ve así: `for (int i = 0; i < calificaciones.Length; i++) { ... calificaciones[i] ... }`. Fíjate en la condición: `i < Length`, con el signo menor estricto, no menor o igual. Si escribieras `i <= calificaciones.Length`, en la última vuelta `i` valdría 5 y `calificaciones[5]` no existe: boom, `IndexOutOfRangeException`. Este es probablemente el error número uno de los principiantes en todo el mundo. Grábalo: para recorrer un arreglo con `for`, la condición es `i < arreglo.Length`.

Además del `for` clásico, C# ofrece el `foreach`, que es aún más simple para recorrer colecciones: `foreach (int cal in calificaciones) { Console.WriteLine(cal); }`. Con `foreach` no te

preocupas por los índices ni por el Length: la variable `cal` toma automáticamente cada valor del arreglo, uno por uno, de principio a fin. Es más legible y elimina por completo el riesgo de equivocarte con los índices. Su limitación es que no puedes modificar los elementos a través de la variable del `foreach` ni saber en qué posición vas; para eso necesitas el `for` clásico. Como regla práctica: si solo necesitas leer los valores, usa `foreach`; si necesitas el índice (por ejemplo, para modificar elementos o comparar posiciones), usa `for`.

Los arreglos también se pueden pasar como parámetros a métodos y devolver como resultado. Por ejemplo, un método que recibe un arreglo de calificaciones y devuelve el promedio. Esto es poderosísimo: un solo método puede procesar colecciones de cualquier tamaño. Cuando pasas un arreglo a un método, no se copia todo el arreglo: se pasa la referencia, es decir, la dirección del arreglo original. Esto significa que si el método modifica los elementos, los cambios se reflejan fuera del método. Para un principiante esto puede ser sorprendente, pero es una característica intencional y muy útil (por ejemplo, un método que ordena un arreglo modifica el original). Solo tenlo presente: los arreglos se comparten, no se clonan, al pasarlos a métodos.

En la vida real, los arreglos están en todas partes: la lista de calificaciones de un grupo, los precios de los productos de una tienda, los nombres de los empleados, las temperaturas registradas cada hora, los puntajes de un videojuego. Cada vez que tengas "muchos datos del mismo tipo", piensa en un arreglo antes de crear variables sueltas.

Índices, Length y recorridos: el trío inseparable

Vamos a profundizar en la relación entre índices, Length y los recorridos, porque dominar este trío es dominar los arreglos. El índice es la posición: un número entero que empieza en 0. Length es el total de elementos. La regla de oro que conecta ambos: el índice válido más grande siempre es `Length - 1`. En un arreglo de 10 elementos, el último índice válido es 9. Si alguna vez dudas, resta uno al tamaño y ese es tu límite.

Hay tres formas principales de recorrer un arreglo en C#. La primera es el `for` con índice, que ya vimos: te da control total, puedes ir hacia adelante o hacia atrás, saltar de dos en dos, modificar elementos. La segunda es el `foreach`: simple y seguro para lectura. La tercera, menos común pero útil, es el `while` con un contador manual. En la práctica profesional usarás `for` y `foreach` el 99 por ciento de las veces.

Un patrón muy común es buscar algo dentro de un arreglo: recorres todos los elementos comparando cada uno con el valor buscado, y cuando lo encuentras guardas el índice o rompes el ciclo. Otro patrón es acumular: sumas todos los elementos para luego calcular un promedio, o cuentas cuántos cumplen una condición. Otro más es transformar: recorres el

arreglo y modificas cada elemento, por ejemplo aplicando un descuento del 10 por ciento a todos los precios. Estos tres patrones (buscar, acumular, transformar) cubren la gran mayoría de lo que harás con arreglos en programas reales, así que practícalos hasta que te salgan sin pensar.

Cuidado con un detalle sutil del foreach: no puedes modificar la colección mientras la recorres con foreach. Si intentas agregar o quitar elementos dentro de un foreach, C# lanza una excepción. Con arreglos de tamaño fijo esto rara vez es un problema, pero es bueno saberlo desde ahora porque cuando llegues a las listas (List<T>) este error aparece con frecuencia.

Otro detalle importante: Length es de solo lectura. No puedes hacer arreglo.Length = 10 para agrandar un arreglo. Los arreglos en C# tienen tamaño fijo: una vez creados con new int[5], siempre tendrán 5 elementos. Si necesitas una colección que crezca, usarás List<int> más adelante en tu carrera. Por ahora, cuando necesites "agrandar" un arreglo, la técnica es crear uno nuevo más grande y copiar los elementos del viejo al nuevo (existe el método Array.Copy que hace exactamente eso, o puedes copiarlos a mano con un ciclo).

Arreglos multidimensionales y matrices: tablas dentro de tu programa

Hasta ahora los arreglos que viste son unidimensionales: una sola fila de casillas, como una lista. Pero ¿qué pasa si necesitas representar una tabla, como el tablero de un gato (tres en raya), una hoja de cálculo, las calificaciones de varios alumnos en varias materias, o los pixeles de una imagen? Para eso existen los arreglos multidimensionales. El más usado es el de dos dimensiones, también llamado matriz: una cuadrícula con filas y columnas.

Piensa en una matriz como un edificio: tiene pisos (filas) y en cada piso hay departamentos numerados (columnas). Para llegar a un departamento necesitas dos datos: el piso y el número de departamento. Igual con la matriz: para llegar a un elemento necesitas dos índices, uno para la fila y otro para la columna. En C# se declara así: int[,] tablero = new int[3, 4];. Esto crea una matriz de 3 filas por 4 columnas, es decir, 12 elementos en total. Los corchetes con la coma [,] indican que es de dos dimensiones. Para acceder a un elemento usas ambos índices separados por coma: tablero[0, 0] es la esquina superior izquierda, tablero[2, 3] es la esquina inferior derecha. Como en los arreglos simples, los índices empiezan en cero: las filas válidas son 0, 1, 2 y las columnas válidas son 0, 1, 2, 3.

También puedes inicializar una matriz con valores directamente: int[,] calificaciones = { { 90, 85, 88 }, { 75, 80, 92 } };. Esto crea una matriz de 2 filas por 3 columnas. La primera fila (índice 0) es { 90, 85, 88 } y la segunda fila (índice 1) es { 75, 80, 92 }. Imagina que las filas son alumnos y las columnas son materias: calificaciones[0, 1] sería la calificación del primer alumno en la segunda materia, que es 85.

Para recorrer una matriz necesitas dos ciclos anidados: uno para las filas y otro para las columnas. El ciclo exterior recorre las filas y el interior recorre las columnas de esa fila. Para saber cuántas filas y columnas tiene la matriz usas el método `GetLength(dimension)`: `matriz.GetLength(0)` devuelve el número de filas y `matriz.GetLength(1)` devuelve el número de columnas. El ciclo típico se ve así: `for (int fila = 0; fila < matriz.GetLength(0); fila++) { for (int col = 0; col < matriz.GetLength(1); col++) { ... matriz[fila, col] ... } }`. Los ciclos anidados son una herramienta fundamental: el interior se ejecuta completo por cada vuelta del exterior. Si la matriz es de 3 por 4, el cuerpo interior se ejecuta 12 veces.

C# también tiene arreglos de más dimensiones (`int[,]` para tres dimensiones, como un cubo), pero en la práctica casi todo se resuelve con una o dos dimensiones. Mención especial merecen los "arreglos dentados" (jagged arrays): `int[][] dentado = new int[3][]`. Esto es un arreglo de arreglos, donde cada fila puede tener diferente longitud: la fila 0 puede tener 2 elementos, la fila 1 puede tener 5, la fila 2 puede tener 3. Se accede con doble corchete: `dentado[1][4]`. Son útiles cuando tus datos no forman un rectángulo perfecto, como las calificaciones de alumnos que cursaron distinto número de materias. Cada sub-arreglo se crea por separado con su propio `new`.

Un error común con matrices es confundir el orden de los índices: `matriz[fila, columna]` y no al revés. Otro es usar `Length` en una matriz esperando el número de filas: `Length` en una matriz devuelve el total de elementos (filas por columnas), no las filas. Para filas y columnas usa siempre `GetLength(0)` y `GetLength(1)`. Y el error clásico se repite aquí en dos dimensiones: índices fuera de rango, ahora con dos coordenadas que vigilar.

Clase 14: Clases y objetos - el salto a la programación orientada a objetos

Hasta este momento del curso, tu código ha sido una secuencia de instrucciones: el programa empieza arriba, ejecuta paso por paso y termina abajo. Los datos vivían en variables sueltas y las operaciones vivían en métodos sueltos. Eso se llama programación estructurada o procedural, y funciona muy bien para programas pequeños. Pero imagina el sistema de un banco: tiene clientes, cuentas, tarjetas, préstamos, sucursales, cajeros, transacciones... Si intentarás modelar todo eso con variables sueltas y métodos sueltos, terminarías con miles de variables llamadas `nombreCliente1`, `saldoCuenta1`, `nombreCliente2`, `saldoCuenta2`... y métodos que reciben veinte parámetros. Sería un caos imposible de mantener. La programación orientada a objetos nació para resolver exactamente ese problema: organizar el código alrededor de las "cosas" que existen en el problema, no alrededor de los pasos del programa.

La idea central de la POO es engañosamente simple: en tu programa creas representaciones de cosas del mundo real (o del mundo del problema) llamadas objetos. Un objeto combina

datos (¿qué sabe?) y comportamientos (¿qué puede hacer?). Un objeto CuentaBancaria sabe su número de cuenta y su saldo (datos), y puede depositar, retirar y consultar su saldo (comportamientos). Un objeto Estudiante sabe su nombre y su matrícula, y puede inscribirse a un curso o consultar sus calificaciones. Los datos se llaman campos o atributos, y los comportamientos se llaman métodos. Cuando juntas datos y comportamientos relacionados en una sola unidad, tu código se vuelve más fácil de entender, de mantener y de reutilizar. Esa unidad es la clase.

Aquí viene la distinción más importante de toda la unidad, así que léela dos veces: la clase es el molde, el objeto es la pieza que sale del molde. La clase Estudiante es la definición general: "todo estudiante tiene un nombre, una matrícula y puede inscribirse". El objeto es un estudiante concreto: "Juan Pérez, matrícula A123, inscrito en C#". Con una sola clase puedes crear miles de objetos, cada uno con sus propios datos pero todos compartiendo la misma estructura y los mismos comportamientos. Sin clases, cada estudiante sería un montón de variables sueltas; con clases, cada estudiante es un objeto ordenado y autocontenido.

La analogía clásica que usan todos los profesores (y que funciona de maravilla) es la del plano de casa. El plano de una casa, dibujado por el arquitecto, no es una casa: no puedes vivir dentro del plano, no tiene paredes reales. El plano dice: "las casas construidas con este plano tendrán 3 recámaras, 2 baños y una cocina, y tendrán un método EncenderLuces". La clase es el plano. Después, la constructora usa ese plano para construir casas reales en diferentes terrenos: la casa de la calle Juárez 123, la casa de la avenida Hidalgo 456. Esas casas sí existen, sí tienen paredes, y cada familia las amuebla diferente. Esas casas son los objetos. Y para construir una casa a partir del plano necesitas a la constructora: en C#, la constructora es la palabra clave new. Cuando escribes Estudiante juan = new Estudiante();, estás diciendo: "constructora, usando el plano Estudiante, constrúyeme una casa (un objeto) y guárdame su dirección en la variable juan".

Vamos a ver la anatomía de una clase en C#. La declaración más simple se ve así: class Estudiante { ... }. Dentro de las llaves van los miembros de la clase. Los campos son variables declaradas dentro de la clase: string nombre; int edad;. Cada objeto creado a partir de la clase tendrá su propia copia de esos campos: el objeto juan tendrá su nombre y su edad, el objeto maria tendrá los suyos, independientes entre sí. Los métodos son funciones declaradas dentro de la clase: void Presentarse() { Console.WriteLine("Hola, soy " + nombre); }. Fíjate que el método usa directamente el campo nombre, sin recibirlo como parámetro: los métodos de un objeto tienen acceso natural a los datos de ese mismo objeto. Eso es parte de la magia de la POO: los datos y el código que los manipula viven juntos.

Los métodos que pertenecen a un objeto (a diferencia de los métodos static que ya conoces de Main) se llaman métodos de instancia, porque operan sobre una instancia específica (un

objeto específico). Cuando llamas `juan.Presentarse()`, el método se ejecuta en el contexto del objeto `juan` y usa los datos de `juan`. Si llamas `maria.Presentarse()`, usa los datos de `maria`. El mismo método, diferentes datos según el objeto. La sintaxis del punto (`objeto.miembro`) es como decir "oye, objeto, haz esto" o "oye, objeto, dame tu dato". Es una conversación entre tu código y los objetos.

Un detalle técnico importante: las clases son tipos de referencia. Esto significa que una variable de tipo `Estudiante` no contiene al objeto directamente, sino la dirección de memoria donde vive el objeto (la referencia). Cuando escribes `Estudiante a = new Estudiante();` `Estudiante b = a;`, no creaste dos estudiantes: creaste un solo objeto y ahora dos variables apuntan a él. Si cambias `b.nombre`, también cambia `a.nombre`, porque es el mismo objeto visto desde dos variables. Esto es igual a lo que vimos con los arreglos (que también son objetos) y es una de las cosas que más sorprenden a los principiantes. No es un error del lenguaje: es así por diseño, y cuando lo entiendes te da mucho poder (pasar objetos a métodos sin copiarlos, por ejemplo).

Otro concepto que debes grabar desde ahora: si declaras `Estudiante juan;` sin el `new`, la variable `juan` vale `null`, que significa "no apunta a ningún objeto". Es como tener la dirección de una casa que nunca se construyó. Si intentas usar `juan.nombre` cuando `juan` es `null`, C# lanza la famosa `NullReferenceException`: "referencia a objeto no establecida como instancia de un objeto". Es el error más común en C# y lo verás muchas veces en tu carrera. La regla es simple: antes de usar un objeto, asegúrate de haberlo creado con `new`. Más adelante en esta unidad veremos este error con más detalle.

¿Por qué todo este esfuerzo? ¿Por qué no seguir con variables sueltas? Por cuatro grandes beneficios que la POO te da y que la industria valora enormemente. Primero, organización: el código agrupado por conceptos es más fácil de navegar; si hay un problema con los estudiantes, sabes que está en la clase `Estudiante`. Segundo, reutilización: una clase bien diseñada se usa en muchos programas sin reescribirla. Tercero, mantenimiento: si cambia una regla del negocio (por ejemplo, cómo se calcula un descuento), la cambias en un solo lugar: el método de la clase. Cuarto, modelado natural: piensas en términos del problema (estudiantes, productos, cuentas) en lugar de términos de la máquina (variables, ciclos, arreglos). Estos beneficios son la razón por la que casi todo el software profesional está escrito con POO.

Campos y métodos de instancia en detalle

Profundicemos en los dos tipos de miembros que verás en esta unidad: campos y métodos de instancia. Un campo es simplemente una variable que vive dentro de la clase y que cada objeto posee por separado. Se declaran como las variables normales, pero a nivel de clase, fuera de cualquier método: `class Producto { string nombre; double precio; int stock; }`. Cada

objeto Producto tendrá su propio nombre, su propio precio y su propio stock. Los campos representan el estado del objeto: cómo está el objeto en este momento. El stock de un producto cambia cuando se vende; el nombre normalmente no.

Los métodos de instancia son los comportamientos del objeto. Se declaran igual que los métodos que ya conoces, pero dentro de la clase y sin la palabra static: `void AplicarDescuento(double porcentaje) { precio = precio - (precio * porcentaje / 100); }`. Este método modifica el campo precio del objeto sobre el que se llama. Si tienes dos productos, laptop y mouse, y llamas `laptop.AplicarDescuento(10)`, solo cambia el precio de la laptop; el mouse ni se entera. Cada objeto tiene sus propios datos, pero todos comparten el mismo código de los métodos. Es como el plano de casa: todas las casas tienen el mismo diseño de interruptores (el método), pero cada familia prende sus propias luces (sus datos).

Dentro de un método de instancia puedes usar los campos directamente por su nombre, porque el método "vive" dentro del objeto y conoce sus datos. También puedes llamar a otros métodos del mismo objeto directamente. Esta cercanía entre datos y código es lo que hace que las clases sean tan expresivas: el método `AplicarDescuento` no necesita que le pases el precio como parámetro, porque el precio ya está ahí, es parte del objeto.

Una convención importante de C#: los campos privados suelen escribirse en camelCase a veces con guion bajo al inicio (`_nombre`), mientras que los métodos y las propiedades usan PascalCase (`AplicarDescuento`). En esta unidad usaremos nombres simples y claros; en la Unidad 6 veremos con más detalle las convenciones de encapsulamiento con `public` y `private`. Por ahora, lo esencial es que practiques crear clases con campos y métodos, crear objetos con `new`, y llamar a sus métodos con la sintaxis del punto.

Un ejercicio mental útil: antes de escribir una clase, pregúntate dos cosas. Uno: ¿qué datos necesita guardar este concepto? Esos serán tus campos. Dos: ¿qué acciones debe poder realizar? Esos serán tus métodos. Para un Estudiante: datos = nombre, matrícula, lista de calificaciones; acciones = inscribirse, agregar calificación, calcular promedio. Para un Producto: datos = nombre, precio, stock; acciones = aplicar descuento, vender (reducir stock), reabastecer. Si puedes responder esas dos preguntas, puedes diseñar cualquier clase básica.

Clase 15: Constructores - el nacimiento de los objetos

Ya sabes crear objetos con `new Estudiante()`. Pero hay un problema: el objeto nace "vacío", con sus campos en valores por defecto (`null`, `0`, `false`). Después tienes que llenarlo a mano, campo por campo: `juan.nombre = "Juan"; juan.matricula = "A123";`. Esto funciona, pero es tedioso y peligroso: ¿qué pasa si olvidas asignar la matrícula? Tendrás un estudiante sin matrícula dando vueltas por tu programa, y el error aparecerá mucho después, lejos de donde

lo causaste. ¿No sería mejor que el objeto naciera ya con sus datos esenciales, como un bebé que nace con nombre? Para eso existen los constructores.

Un constructor es un método especial que se ejecuta automáticamente en el momento en que el objeto nace, es decir, cuando se ejecuta el `new`. Su trabajo es dejar al objeto en un estado inicial válido: asignar los valores iniciales a los campos. Se reconoce por dos reglas: se llama exactamente igual que la clase y no declara tipo de retorno (ni siquiera `void`). Ejemplo: `class Estudiante { string nombre; string matricula; public Estudiante(string nombre, string matricula) { this.nombre = nombre; this.matricula = matricula; } }`. Ahora puedes crear el objeto ya con datos: `Estudiante juan = new Estudiante("Juan Pérez", "A123");`. En una sola línea el objeto nace y queda inicializado. No hay forma de olvidar la matrícula, porque el constructor la exige.

Aquí aparece la palabra clave `this`, que merece su propia explicación. Dentro del constructor (o de cualquier método de instancia), `this` significa "este objeto, el que se está creando o el que está ejecutando el método". En el ejemplo anterior, el parámetro se llama `nombre` y el campo también se llama `nombre`. ¿Cómo distingue C# entre ambos? Con `this`: `this.nombre` se refiere al campo del objeto, mientras que `nombre` a secas se refiere al parámetro (la variable más cercana). Es una convención muy común llamar a los parámetros igual que los campos y usar `this` para desambiguar. Además de desambiguar, `this` sirve para pasar el objeto actual como argumento a otro método o para encadenar constructores, aunque eso lo verás más adelante. Por ahora quédate con: `this` = "este objeto".

Una clase puede tener varios constructores con diferentes parámetros. Esto se llama sobrecarga de constructores y es muy útil: puedes ofrecer varias formas de nacer. Por ejemplo, la clase `Producto` podría tener un constructor que recibe `nombre` y `precio`, y otro que además recibe el `stock`: `public Producto(string nombre, double precio) { ... stock = 0; }` y `public Producto(string nombre, double precio, int stock) { ... }`. Así, quien crea el objeto elige la versión que le convenga: `new Producto("Laptop", 15000)` o `new Producto("Mouse", 300, 50)`. C# elige automáticamente el constructor correcto según los argumentos que le pases. Es como tener varias puertas de entrada al mismo edificio.

¿Y qué pasa si no escribes ningún constructor? C# te regala uno automáticamente: el constructor por defecto (también llamado constructor sin parámetros o implícito). Es un constructor vacío, sin parámetros, que simplemente crea el objeto con valores por defecto. Por eso `new Estudiante()` funcionaba aunque nunca escribiste un constructor: C# lo puso por ti, en silencio. Pero atención, hay una trampa importante: en cuanto escribes TU propio constructor con parámetros, C# retira el regalo. El constructor por defecto desaparece. Si tu clase solo tiene `public Estudiante(string nombre, string matricula)`, entonces `new Estudiante()` sin argumentos ya no compila: C# te dirá que no existe un constructor sin parámetros. Este es un error de compilación muy común entre principiantes. La solución es simple: si necesitas ambas formas, escribe ambos constructores explícitamente.

Los constructores también son el lugar ideal para validar los datos iniciales. Por ejemplo, en el constructor de `Producto` puedes verificar que el precio no sea negativo: si alguien intenta crear `new Producto("Laptop", -500)`, el constructor puede lanzar una excepción o ajustar el valor. De esta forma garantizas que ningún objeto nazca en un estado inválido. Esta idea (validar al nacer) es una de las bases del buen diseño orientado a objetos y la retomaremos con más fuerza en la Unidad 6 cuando hablemos de encapsulamiento.

En resumen: los constructores garantizan que todo objeto nazca bien formado, con sus datos esenciales. Úsalos siempre que una clase tenga datos obligatorios. Un objeto que nace válido te ahorra una cantidad enorme de errores difíciles de rastrear.

Errores comunes y cómo evitarlos

Esta sección es oro puro: los errores que verás a continuación los comete todo principiante (y a veces hasta los programadores con experiencia en un mal día). Conocerlos de antemano te ahorrará horas de frustración.

El error número uno con arreglos es el índice fuera de rango: `IndexOutOfRangeException`. Ocurre cuando intentas acceder a una posición que no existe: un arreglo de 5 elementos y pides el índice 5, o un índice negativo como -1. El mensaje de la excepción te dice exactamente qué pasó, pero el programa se detiene. ¿Cómo evitarlo? Tres reglas: uno, recuerda que los índices van de 0 a `Length - 1`. Dos, en los ciclos `for` usa siempre `i < arreglo.Length` (menor estricto, nunca menor o igual). Tres, si el índice viene de un cálculo o de la entrada del usuario, válidalo antes de usarlo: `if (indice >= 0 && indice < arreglo.Length)`. Un caso típico: pedirle al usuario un número de opción del 1 al 5 y usarlo directo como índice sin restar 1. El usuario piensa en base 1, tu arreglo en base 0: convierte siempre.

El error número dos, el más famoso de C#, es la `NullReferenceException`: intentar usar un objeto que es `null`, es decir, que nunca fue creado con `new`. Los escenarios típicos: declaras

Estudiante juan; y luego intentas `juan.nombre = "Juan"` sin haber hecho `new Estudiante()`; declaras un arreglo `int[] numeros`; sin el `new int[...]`; o un método recibe un objeto como parámetro y alguien le pasa `null`. El mensaje "referencia a objeto no establecida como instancia de un objeto" significa exactamente eso: intentaste usar algo que no existe. ¿Cómo evitarlo? Asegúrate de que cada variable de objeto tenga su `new` antes de usarse. Si un método puede recibir `null`, válidalo al inicio. Y cuando veas esta excepción, la pregunta que debes hacerte es siempre la misma: "¿en qué línea, qué objeto estoy usando que podría no haber sido creado?".

El error número tres es conceptual: confundir la clase con el objeto. Se manifiesta de varias formas. Una es intentar usar la clase como si fuera un objeto: `Estudiante.nombre = "Juan"`; sin haber creado ningún estudiante. La clase es el plano, no la casa: no puedes cambiarle el color a "el plano", solo a una casa construida. Otra manifestación es crear un solo objeto y reusarlo cuando necesitabas varios: en un ciclo que lee 5 estudiantes, crear el objeto una vez fuera del ciclo y solo cambiarle los datos dentro: al final tendrás 5 variables apuntando al mismo objeto con los datos del último. La solución: el `new` va dentro del ciclo, para que cada vuelta nazca un objeto nuevo e independiente.

El error número cuatro es olvidar que los arreglos y objetos se pasan por referencia. Si escribes un método que "ordena una copia" del arreglo pero en realidad modifica el original, tendrás un bug sutil. Si necesitas una copia real, créala explícitamente (con un ciclo o con métodos de copiado). Y al revés: si esperabas que el método modificara tu arreglo y no lo hizo porque trabajaste sobre una copia, también hay bug. Ten siempre claro si estás trabajando con el original o con una copia.

El error número cinco es específico de constructores: escribir un constructor con parámetros y luego intentar usar `new SinParametros()`, olvidando que el constructor por defecto desapareció. El compilador te avisa con un error claro, así que no es grave, pero confunde a los principiantes porque "antes funcionaba". Recuerda: si defines cualquier constructor, define también el sin parámetros si lo vas a necesitar.

Buenas prácticas profesionales

Más allá de evitar errores, hay hábitos que distinguen al programador profesional del aficionado. Primero: usa `foreach` cuando solo necesites leer. Es más legible, más corto y elimina los errores de índices. Reserva el `for` con índice para cuando necesites modificar elementos o conocer la posición.

Segundo: nunca uses números mágicos como límites de ciclos. En lugar de `for (int i = 0; i < 5; i++)`, usa `for (int i = 0; i < arreglo.Length; i++)`. Si mañana el arreglo cambia de tamaño, tu ciclo sigue funcionando. Los números mágicos son una deuda que siempre se cobra.

Tercero: inicializa tus objetos con constructores que exijan los datos esenciales. Si un `Estudiante` no tiene sentido sin matrícula, el constructor debe pedirla. Así haces imposible (o al menos muy difícil) crear objetos inválidos. Un objeto que nace válido es un objeto que no te dará sorpresas.

Cuarto: dale a tus clases una sola responsabilidad clara. Una clase `Estudiante` debe encargarse de cosas de estudiantes, no de imprimir reportes en PDF ni de conectarse a la base de datos. Si una clase hace demasiadas cosas, divídela. Este principio tiene nombre elegante (principio de responsabilidad única) pero la idea es de sentido común: cada cosa en su lugar.

Quinto: nombra con claridad. Una clase se nombra con un sustantivo en singular y PascalCase: `Estudiante`, `Producto`, `CuentaBancaria`. Los métodos con verbos: `CalcularPromedio`, `AplicarDescuento`, `Inscribir`. Los campos con nombres descriptivos: `nombre`, `precio`, `stock`. El código se lee muchas más veces de las que se escribe: invierte en buenos nombres.

Sexto: valida las entradas en los límites de tu código. Si un método recibe un índice, verifica que esté en rango. Si un constructor recibe un precio, verifica que no sea negativo. Es mejor fallar rápido con un mensaje claro que dejar que un dato inválido viaje por el programa y cause un error misterioso diez pasos después.

Ejemplos paso a paso

Ejemplo 1: Calificaciones de un grupo con arreglos

Vamos a construir un programa completo que usa un arreglo para guardar las calificaciones de 5 estudiantes, calcula el promedio, encuentra la calificación más alta y cuenta cuántos aprobaron (calificación mayor o igual a 60). Este ejemplo junta todo lo de la clase 13: declaración, llenado, `Length`, recorridos con `for` y `foreach`, y los patrones de acumular y buscar.

```
using System;

class Program
{
    static void Main()
    {
```

```

// Creamos un arreglo de 5 enteros. new reserva el espacio.
// Cada posición inicia en 0 automáticamente.
int[] calificaciones = new int[5];

// Llenamos el arreglo pidiendo datos al usuario.
// El ciclo va de i = 0 hasta i < 5 (Length), es decir, 0,1,2,3,4.
for (int i = 0; i < calificaciones.Length; i++)
{
    Console.Write("Calificación del estudiante " + (i + 1) + ": ");
    // i + 1 es solo para mostrarle al usuario números del 1 al 5,
    // pero el índice real que usamos sigue siendo i (base 0).
    calificaciones[i] = int.Parse(Console.ReadLine());
}

// Patrón ACUMULAR: sumamos todos los elementos para el promedio.
int suma = 0;
foreach (int cal in calificaciones)
{
    suma = suma + cal; // o suma += cal;
}
// Convertimos a double antes de dividir para no perder decimales.
double promedio = (double)suma / calificaciones.Length;

// Patrón BUSCAR EL MÁXIMO: empezamos suponiendo que el
// primero es el mayor y comparamos con los demás.
int maxima = calificaciones[0];
for (int i = 1; i < calificaciones.Length; i++)
{
    if (calificaciones[i] > maxima)
    {
        maxima = calificaciones[i];
    }
}

// Patrón CONTAR: contamos cuántos cumplen una condición.
int aprobados = 0;
foreach (int cal in calificaciones)
{
    if (cal >= 60)
    {
        aprobados++;
    }
}

Console.WriteLine("Promedio del grupo: " + promedio);

```

```

        Console.WriteLine("Calificación más alta: " + maxima);
        Console.WriteLine("Estudiantes aprobados: " + aprobados);
    }
}

```

Explicación línea por línea: empezamos con `int[] calificaciones = new int[5];` que crea el "librero" con 5 casillas, todas en cero. El primer `for` llena el arreglo: fíjate que la condición es `i < calificaciones.Length`, el patrón correcto que evita el índice fuera de rango. Dentro del ciclo mostramos `(i + 1)` al usuario porque las personas cuentan desde 1, pero guardamos en `calificaciones[i]` porque el arreglo cuenta desde 0. Esta conversión entre "número humano" y "índice real" es un patrón que usarás constantemente.

Después calculamos la suma con `foreach`, que es ideal aquí porque solo leemos valores y no necesitamos los índices. El cast `(double)suma` es crucial: sin él, la división de dos enteros en C# trunca los decimales (`7 / 2` da 3, no 3.5). Para el máximo, inicializamos `maxima` con el primer elemento (no con cero, porque todas las calificaciones podrían ser negativas en otros contextos, o podrías estar buscando el mínimo) y comparamos desde el índice 1. Para contar aprobados usamos otro `foreach` con un contador que aumenta cuando se cumple la condición. Tres patrones fundamentales en un solo programa: acumular, buscar el máximo y contar con condición.

Ejemplo 2: Matriz de calificaciones por alumno y materia

Ahora subimos de nivel: una matriz de 3 alumnos por 4 materias. Llenaremos la matriz, la imprimiremos como tabla y calcularemos el promedio de cada alumno (promedio por fila). Este ejemplo practica la declaración de matrices, los índices dobles, `GetLength` y los ciclos anidados.

```

using System;

class Program
{
    static void Main()
    {
        // Matriz de 3 filas (alumnos) x 4 columnas (materias).
        // La coma dentro de los corchetes [,] indica dos dimensiones.
        int[,] califs = new int[3, 4];

        // Llenado con ciclos anidados: el exterior recorre filas,
        // el interior recorre las columnas de cada fila.
        for (int fila = 0; fila < califs.GetLength(0); fila++)
        {
            for (int col = 0; col < califs.GetLength(1); col++)

```

```

        {
            Console.Write("Alumno " + (fila + 1) + ", materia " + (col + 1) + ":
");
            califs[fila, col] = int.Parse(Console.ReadLine());
        }
    }

    Console.WriteLine("\nTabla de calificaciones:");
    // Imprimimos la matriz como tabla, fila por fila.
    for (int fila = 0; fila < califs.GetLength(0); fila++)
    {
        for (int col = 0; col < califs.GetLength(1); col++)
        {
            // \t es un tabulador: separa las columnas visualmente.
            Console.Write(califs[fila, col] + "\t");
        }
        // Al terminar una fila, salto de línea para empezar la siguiente.
        Console.WriteLine();
    }

    // Promedio por alumno: para cada fila sumamos sus columnas
    // y dividimos entre el número de columnas.
    Console.WriteLine("\nPromedios por alumno:");
    for (int fila = 0; fila < califs.GetLength(0); fila++)
    {
        int suma = 0;
        for (int col = 0; col < califs.GetLength(1); col++)
        {
            suma += califs[fila, col];
        }
        double promedio = (double)suma / califs.GetLength(1);
        Console.WriteLine("Alumno " + (fila + 1) + ": " + promedio);
    }
}

```

Explicación: la declaración `int[,] califs = new int[3, 4];` crea 12 casillas organizadas en 3 filas y 4 columnas. `GetLength(0)` devuelve 3 (filas) y `GetLength(1)` devuelve 4 (columnas): el 0 y el 1 se refieren a la primera y segunda dimensión. En el llenado, el ciclo exterior fija la fila y el interior recorre todas sus columnas antes de pasar a la siguiente fila; esto se llama orden por filas y es la forma natural de llenar una matriz. Al imprimir, usamos `Console.Write` (sin salto de línea) dentro del ciclo interior para que las columnas queden en el mismo renglón, y `Console.WriteLine()` al final de cada fila para saltar al siguiente renglón. El tabulador `"\t"` alinea las columnas. Para el promedio por alumno, la suma se reinicia a cero en cada fila (fíjate que

int suma = 0; está dentro del ciclo exterior): un error típico es declarar la suma fuera y que se acumulen los datos de todos los alumnos. Dividimos entre GetLength(1) porque cada alumno tiene tantas materias como columnas.

Ejemplo 3: Nuestra primera clase - Estudiante

Es momento del salto a la POO. Vamos a crear la clase Estudiante con campos (nombre, matrícula, arreglo de calificaciones) y métodos (agregar calificación, calcular promedio, presentarse). Luego la usaremos desde Main creando objetos con new.

```
using System;

// La clase es el plano: define qué datos y qué comportamientos
// tendrá cada estudiante, pero aún no es ningún estudiante concreto.
class Estudiante
{
    // CAMPOS: los datos que cada objeto guardará por separado.
    string nombre;
    string matricula;
    // Cada estudiante tiene su propio arreglo de 5 calificaciones.
    int[] calificaciones = new int[5];
    // Contador de cuántas calificaciones se han guardado realmente.
    int totalCalifs = 0;

    // MÉTODOS DE INSTANCIA: comportamientos del objeto.
    // Guardan o usan los datos del objeto que los llama.

    // Asigna los datos básicos del estudiante.
    public void Registrar(string nom, string mat)
    {
        nombre = nom;
        matricula = mat;
    }

    // Agrega una calificación al arreglo, si hay espacio.
    public void AgregarCalificacion(int cal)
    {
        if (totalCalifs < calificaciones.Length)
        {
            calificaciones[totalCalifs] = cal;
            totalCalifs++; // avanzamos a la siguiente casilla libre
        }
        else
        {

```

```

        Console.WriteLine("Ya no hay espacio para más calificaciones.");
    }
}

// Calcula el promedio solo de las calificaciones guardadas.
public double CalcularPromedio()
{
    if (totalCalifs == 0)
    {
        return 0; // evitamos dividir entre cero
    }
    int suma = 0;
    // Solo recorreremos hasta totalCalifs, no todo el arreglo,
    // porque las casillas vacías aún tienen ceros que contaminarían el promedio.
    for (int i = 0; i < totalCalifs; i++)
    {
        suma += calificaciones[i];
    }
    return (double)suma / totalCalifs;
}

// Muestra la información del estudiante.
public void Presentarse()
{
    Console.WriteLine("Soy " + nombre + ", matrícula " + matricula +
        ", promedio: " + CalcularPromedio());
}
}

class Program
{
    static void Main()
    {
        // Creamos OBJETOS a partir de la clase (el plano).
        // Cada new construye un estudiante independiente.
        Estudiante juan = new Estudiante();
        Estudiante maria = new Estudiante();

        // Llenamos los datos de cada objeto con sus métodos.
        juan.Registrar("Juan Pérez", "A001");
        juan.AgregarCalificacion(90);
        juan.AgregarCalificacion(85);
        juan.AgregarCalificacion(92);

        maria.Registrar("María López", "A002");
    }
}

```



```

        maria.AgregarCalificacion(100);
        maria.AgregarCalificacion(98);

        // Cada objeto responde con SUS propios datos.
        juan.Presentarse(); // Soy Juan Pérez, matrícula A001, promedio: 89
        maria.Presentarse(); // Soy María López, matrícula A002, promedio: 99

        // juan y maria son independientes: lo que le pasa a uno
        // no afecta al otro. Esa es la gracia de los objetos.
    }
}

```

Explicación línea por línea: la clase Estudiante declara tres campos: nombre, matricula y un arreglo calificaciones de 5 posiciones que se crea con new directamente en la declaración del campo (esto es válido y muy común: cada objeto nace con su arreglo ya creado). El campo totalCalifs lleva la cuenta de cuántas calificaciones reales hay, porque el arreglo siempre tiene 5 casillas pero quizá solo se llenaron 3; las demás tienen ceros automáticos que no deben entrar al promedio. El método Registrar asigna nombre y matrícula. AgregarCalificacion verifica que haya espacio antes de guardar (buena práctica: validar antes de actuar) y avanza el contador. CalcularPromedio primero se protege contra la división entre cero (si no hay calificaciones, devuelve 0) y luego suma solo las posiciones ocupadas. Presentarse combina todo y además llama a otro método del mismo objeto (CalcularPromedio) directamente, sin punto, porque ya está "dentro" del objeto.

En Main, new Estudiante() construye cada objeto: juan y maria son dos casas construidas con el mismo plano pero amuebladas diferente. La sintaxis juan.Registrar(...) significa "oye juan, regístrate con estos datos". Cada llamada a Presentarse usa los datos del objeto que la llamó. Si agregas una calificación a juan, el promedio de maria no cambia: independencia total entre objetos.

Ejemplo 4: Clase Producto con constructores y arreglo de precios

Ahora combinamos arreglos y POO con constructores. La clase Producto tendrá un constructor con parámetros que exige nombre y precio, un arreglo con el historial de precios, y métodos para aplicar descuento y mostrar el historial. Veremos también el constructor sin parámetros y la palabra this.

```

using System;

class Producto
{
    string nombre;

```

```

double precio;
int stock;
// Historial de los últimos 10 precios del producto.
double[] historialPrecios = new double[10];
int totalPrecios = 0;

// CONSTRUCTOR CON PARÁMETROS: el objeto nace ya con datos.
// this.nombre se refiere al campo; nombre (a secas) al parámetro.
public Producto(string nombre, double precio)
{
    this.nombre = nombre;
    // Validamos al nacer: un precio negativo no tiene sentido.
    if (precio < 0)
    {
        this.precio = 0;
    }
    else
    {
        this.precio = precio;
    }
    this.stock = 0;
    // Guardamos el precio inicial como primer dato del historial.
    RegistrarPrecio(this.precio);
}

// CONSTRUCTOR SIN PARÁMETROS: nace con valores neutros.
// Necesario porque al definir el constructor de arriba,
// C# ya no nos regala el constructor por defecto.
public Producto()
{
    this.nombre = "Sin nombre";
    this.precio = 0;
    this.stock = 0;
}

// Guarda un precio en el historial si hay espacio.
public void RegistrarPrecio(double nuevoPrecio)
{
    if (totalPrecios < historialPrecios.Length)
    {
        historialPrecios[totalPrecios] = nuevoPrecio;
        totalPrecios++;
    }
}

```

```

// Aplica un descuento y registra el nuevo precio en el historial.
public void AplicarDescuento(double porcentaje)
{
    if (porcentaje < 0 || porcentaje > 100)
    {
        Console.WriteLine("Porcentaje inválido.");
        return;
    }
    precio = precio - (precio * porcentaje / 100);
    RegistrarPrecio(precio);
}

// Suma unidades al stock (reabastecer).
public void Reabastecer(int cantidad)
{
    if (cantidad > 0)
    {
        stock += cantidad;
    }
}

// Muestra la ficha completa del producto.
public void MostrarFicha()
{
    Console.WriteLine("Producto: " + nombre);
    Console.WriteLine("Precio actual: $" + precio);
    Console.WriteLine("Stock: " + stock + " unidades");
    Console.WriteLine("Historial de precios:");
    for (int i = 0; i < totalPrecios; i++)
    {
        Console.WriteLine("  - $" + historialPrecios[i]);
    }
}

}

class Program
{
    static void Main()
    {
        // Nace con datos gracias al constructor con parámetros.
        Producto laptop = new Producto("Laptop Gamer", 25000);
        laptop.Reabastecer(10);
        laptop.AplicarDescuento(10); // 10% de descuento: 25000 -> 22500
        laptop.AplicarDescuento(5); // 5% sobre 22500 -> 21375
        laptop.MostrarFicha();
    }
}

```

```

        Console.WriteLine();

        // Nace con valores neutros gracias al constructor sin parámetros.
        Producto generico = new Producto();
        generico.MostrarFicha();
    }
}

```

Explicación: el constructor `public Producto(string nombre, double precio)` se ejecuta automáticamente con cada `new Producto("Laptop Gamer", 25000)`. Dentro, `this.nombre` distingue el campo del parámetro con el mismo nombre. La validación `if (precio < 0)` garantiza que ningún producto nazca con precio negativo: el objeto nace válido o no nace. El constructor sin parámetros existe porque lo escribimos explícitamente; sin él, `new Producto()` no compilaría. El historial de precios es un arreglo que crece con cada cambio de precio gracias a `RegistrarPrecio`, que usa el patrón del contador (`totalPrecios`) para saber dónde guardar. `AplicarDescuento` valida el porcentaje (entre 0 y 100), modifica el precio y registra el cambio: el objeto se encarga solo de mantener su historial coherente. En Main vemos las dos formas de nacer y cómo cada objeto mantiene su propio estado: la laptop tiene su historial de tres precios, el producto genérico tiene valores neutros.

Ejemplo 5: Arreglo de objetos - la tienda completa

El ejemplo final combina todo: un arreglo donde cada elemento es un objeto `Producto`. Así se modela una tienda real: una colección de productos, cada uno con sus datos y comportamientos. Veremos cómo crear el arreglo, llenarlo con objetos (¡cada posición necesita su propio `new`!) y recorrerlo.

```

using System;

class Producto
{
    string nombre;
    double precio;

    public Producto(string nombre, double precio)
    {
        this.nombre = nombre;
        this.precio = precio;
    }

    public double ObtenerPrecio()
    {

```

```

        return precio;
    }

    public void Mostrar()
    {
        Console.WriteLine(nombre + " - $" + precio);
    }
}

class Program
{
    static void Main()
    {
        // Arreglo de 3 PRODUCTOS (no de números, no de textos:
        // cada casilla guardará un objeto Producto).
        Producto[] catalogo = new Producto[3];

        // ¡OJO! new Producto[3] solo crea el arreglo (el librero),
        // pero cada casilla está en null. Hay que construir
        // cada objeto por separado:
        catalogo[0] = new Producto("Laptop", 25000);
        catalogo[1] = new Producto("Mouse", 350);
        catalogo[2] = new Producto("Teclado", 900);

        // Recorremos el catálogo con foreach: cada producto
        // se muestra a sí mismo con su propio método.
        Console.WriteLine("Catálogo de la tienda:");
        foreach (Producto p in catalogo)
        {
            p.Mostrar();
        }

        // Calculamos el valor total del inventario sumando
        // el precio de cada objeto (patrón acumular, con objetos).
        double total = 0;
        foreach (Producto p in catalogo)
        {
            total += p.ObtenerPrecio();
        }
        Console.WriteLine("Valor total del inventario: $" + total);
    }
}

```

Explicación: `Producto[] catalogo = new Producto[3];` crea un arreglo capaz de guardar 3 referencias a `Producto`, pero las tres casillas inician en `null`: el librero existe pero está vacío.

Cada `catalogo[i] = new Producto(...)`; construye un objeto real y guarda su dirección en la casilla. Si olvidaras estos `new` e intentaras `catalogo[0].Mostrar()`, obtendrías una `NullPointerException`, porque estarías llamando a un método sobre `null`. Este es uno de los errores más comunes al trabajar con arreglos de objetos, así que memorízalo: el `new` del arreglo no crea los objetos, solo el contenedor. Después, recorrer el arreglo con `foreach` nos da cada objeto en la variable `p`, y `p.Mostrar()` le pide a cada producto que se muestre a sí mismo. Para el total usamos el patrón acumular, pero en lugar de sumar números directos, le preguntamos a cada objeto su precio con `p.ObtenerPrecio()`. Fíjate en la elegancia: `Main` no necesita saber cómo el producto guarda su precio; solo le pide el dato. Esa separación es la esencia de la POO.

Práctica

Resuelve cada ejercicio por tu cuenta antes de mirar la solución. Escribe el código, ejecútalo, experimenta con él y solo después compara con la solución propuesta. Equivocarse aquí, en la práctica, es gratis: equivócate todo lo que necesites.

1. Temperaturas de la semana. Crea un arreglo de 7 valores `double` con las temperaturas registradas cada día de la semana (inventa los valores al declararlo). Recorre el arreglo para calcular e imprimir: la temperatura promedio, la más alta y la más baja. Usa `foreach` donde sea posible y `for` con índice donde lo necesites.
2. El buscador. Escribe un programa que tenga un arreglo de 10 enteros con valores inventados por ti. Pídele al usuario un número y busca si ese número existe en el arreglo. Si existe, imprime en qué posición (índice) lo encontraste; si no existe, imprime que no se encontró. Pista: usa una variable booleana como bandera y rompe el ciclo con `break` cuando lo encuentres.
3. Tabla de multiplicar en matriz. Crea una matriz de 10 x 10 donde la casilla `[fila, columna]` contenga el resultado de $(\text{fila} + 1) * (\text{columna} + 1)$. Es decir, la tabla de multiplicar del 1 al 10. Imprímela como tabla bien alineada usando tabuladores. Pista: necesitas dos ciclos anidados y no necesitas pedir datos al usuario, los calculas.
4. Clase Libro. Diseña una clase `Libro` con los campos `titulo`, `autor` y `paginas`, un constructor con parámetros que reciba los tres datos (validando que `paginas` no sea negativo), y un método `MostrarFicha` que imprima la información. En `Main` crea dos objetos `Libro` diferentes, muestra sus fichas y verifica que cada objeto conserve sus propios datos.
5. Clase CuentaBancaria. Crea una clase `CuentaBancaria` con campos `numeroCuenta` y `saldo`, un constructor que reciba el número de cuenta e inicie el saldo en cero, y los métodos `Depositar(cantidad)`, `Retirar(cantidad)` y `ConsultarSaldo()`. `Depositar` debe ignorar

cantidades negativas o cero. Retirar debe impedir que el saldo quede negativo (si no hay fondos suficientes, muestra un mensaje y no hace el retiro). En Main crea una cuenta, deposita 1000, retira 300, intenta retirar 1000 (debe fallar) y consulta el saldo final.

6. Inventario con arreglo de objetos. Combina lo aprendido: crea una clase Producto sencilla (nombre, precio, stock, constructor con parámetros y método Mostrar). En Main crea un arreglo de 4 productos, llénalo con new en cada posición, recórrelo con foreach para mostrar todos los productos, y calcula el valor total del inventario multiplicando precio por stock de cada producto y sumando.
7. Reto: tres en raya. Crea una matriz char de 3 x 3 que represente un tablero de gato. Inicialízala con espacios en blanco usando ciclos anidados. Coloca una 'X' en el centro ([1,1]) y un 'O' en la esquina superior izquierda ([0,0]). Imprime el tablero con formato, con líneas separadoras entre filas para que se vea como un gato de verdad. Este ejercicio practica inicialización de matrices, índices dobles e impresión con formato.
8. Reto: clase Rectangulo con validación. Diseña una clase Rectangulo con campos baseRect y altura, un constructor con parámetros que solo acepte valores positivos (si recibe un valor no positivo, lo ajusta a 1), y los métodos CalcularArea() y CalcularPerimetro(). En Main crea tres rectángulos (uno con datos normales, uno con un valor negativo para probar la validación, uno con el constructor sin parámetros si lo agregaste) e imprime área y perímetro de cada uno.

Soluciones

Solución 1: temperaturas de la semana. La clave está en inicializar el arreglo con valores directos y usar los patrones de acumular y buscar máximo/mínimo. Para el mínimo, el truco es inicializar con el primer elemento igual que hicimos con el máximo.

```

using System;

class Program
{
    static void Main()
    {
        // Arreglo inicializado con valores directos: C# calcula el tamaño solo.
        double[] temperaturas = { 22.5, 24.0, 19.8, 21.3, 25.6, 23.1, 20.4 };

        double suma = 0;
        double maxima = temperaturas[0]; // empezamos con el primero
        double minima = temperaturas[0]; // igual para el mínimo

        foreach (double t in temperaturas)
        {
            suma += t;
            if (t > maxima)
            {
                maxima = t;
            }
            if (t < minima)
            {
                minima = t;
            }
        }

        double promedio = suma / temperaturas.Length;

        Console.WriteLine("Promedio: " + promedio);
        Console.WriteLine("Máxima: " + maxima);
        Console.WriteLine("Mínima: " + minima);
    }
}

```

Explicación: usamos foreach porque solo leemos valores y no necesitamos los índices. Inicializar maxima y minima con temperaturas[0] (en lugar de 0) es importante: si todas las temperaturas fueran negativas, inicializar maxima en 0 daría un resultado incorrecto. El promedio se calcula dividiendo la suma entre Length, que aquí vale 7. Todo el recorrido se hace en una sola pasada por el arreglo: eficiente y legible.

Solución 2: el buscador. El patrón de búsqueda lineal: recorrer, comparar, y usar una bandera para recordar si se encontró.


```

using System;

class Program
{
    static void Main()
    {
        int[] numeros = { 4, 18, 7, 23, 9, 31, 12, 7, 40, 15 };

        Console.Write("¿Qué número buscas? ");
        int buscado = int.Parse(Console.ReadLine());

        bool encontrado = false; // bandera: aún no lo encontramos
        int posicion = -1;        // -1 significa "no existe" por convención

        for (int i = 0; i < numeros.Length; i++)
        {
            if (numeros[i] == buscado)
            {
                encontrado = true;
                posicion = i;
                break; // ya lo encontramos, no seguimos buscando
            }
        }

        if (encontrado)
        {
            Console.WriteLine("Encontrado en la posición " + posicion);
        }
        else
        {
            Console.WriteLine("El número no está en el arreglo.");
        }
    }
}

```

Explicación: la variable booleana encontrado empieza en false y solo cambia a true si vemos el valor. El break detiene el ciclo en cuanto encontramos el número: no tiene sentido seguir buscando. Fíjate que usamos for con índice y no foreach, porque necesitamos saber la posición donde lo encontramos. La convención de usar -1 para "no encontrado" es muy común en programación (muchos métodos de búsqueda de la vida real devuelven -1 en ese caso). Nota que el número 7 aparece dos veces en el arreglo: nuestro programa reporta la primera aparición, que es el comportamiento estándar de la búsqueda lineal.

Solución 3: tabla de multiplicar en matriz. Aquí no pedimos datos: cada casilla se calcula con la fórmula.

```
using System;

class Program
{
    static void Main()
    {
        int[,] tabla = new int[10, 10];

        // Llenado: la casilla [f, c] guarda (f+1) * (c+1).
        // Sumamos 1 porque los índices empiezan en 0 pero la
        // tabla de multiplicar empieza en 1.
        for (int f = 0; f < tabla.GetLength(0); f++)
        {
            for (int c = 0; c < tabla.GetLength(1); c++)
            {
                tabla[f, c] = (f + 1) * (c + 1);
            }
        }

        // Impresión como tabla alineada.
        for (int f = 0; f < tabla.GetLength(0); f++)
        {
            for (int c = 0; c < tabla.GetLength(1); c++)
            {
                Console.Write(tabla[f, c] + "\t");
            }
            Console.WriteLine();
        }
    }
}
```

Explicación: la fórmula $(f + 1) * (c + 1)$ convierte los índices base 0 en factores base 1: la casilla [0,0] guarda $1*1=1$, la [2,3] guarda $3*4=12$. Este es un ejemplo perfecto de matriz donde los datos se derivan de la posición. Los dos ciclos anidados recorren las 100 casillas. Al imprimir, el tabulador mantiene las columnas alineadas aunque los números tengan uno o dos dígitos (1 al 100).

Solución 4: clase Libro. Nuestra primera clase diseñada desde cero con constructor y validación.

```

using System;

class Libro
{
    string titulo;
    string autor;
    int paginas;

    // Constructor con parámetros: el libro nace con sus datos.
    public Libro(string titulo, string autor, int paginas)
    {
        this.titulo = titulo;
        this.autor = autor;
        // Validación al nacer: páginas negativas no tienen sentido.
        if (paginas < 0)
        {
            this.paginas = 0;
        }
        else
        {
            this.paginas = paginas;
        }
    }

    public void MostrarFicha()
    {
        Console.WriteLine("\"" + titulo + "\" de " + autor + ", " + paginas + "
páginas.");
    }
}

class Program
{
    static void Main()
    {
        Libro libro1 = new Libro("Cien años de soledad", "Gabriel García Márquez",
432);
        Libro libro2 = new Libro("El principito", "Antoine de Saint-Exupéry", 96);

        libro1.MostrarFicha();
        libro2.MostrarFicha();
        // Cada objeto muestra SUS datos: son independientes.
    }
}

```

Explicación: el constructor exige los tres datos, así que es imposible crear un Libro sin título, sin autor o sin páginas: el objeto nace completo. La validación de paginas protege contra datos absurdos. En Main, libro1 y libro2 son dos objetos independientes construidos con el mismo plano; cada uno guarda sus propios valores en sus propios campos. Observa cómo el método MostrarFicha accede directamente a los campos: está dentro del objeto y conoce sus datos.

Solución 5: clase CuentaBancaria. El ejemplo clásico de encapsulamiento básico: el saldo solo cambia a través de métodos que aplican reglas.

```
using System;

class CuentaBancaria
{
    string numeroCuenta;
    double saldo;

    // El constructor pide el número de cuenta; el saldo nace en cero.
    public CuentaBancaria(string numeroCuenta)
    {
        this.numeroCuenta = numeroCuenta;
        this.saldo = 0;
    }

    public void Depositar(double cantidad)
    {
        // Regla: no se aceptan depósitos de cero o negativos.
        if (cantidad > 0)
        {
            saldo += cantidad;
            Console.WriteLine("Depósito exitoso. Saldo: $" + saldo);
        }
        else
        {
            Console.WriteLine("Cantidad inválida para depósito.");
        }
    }

    public void Retirar(double cantidad)
    {
        // Regla 1: la cantidad debe ser positiva.
        // Regla 2: debe haber fondos suficientes.
        if (cantidad <= 0)
        {
            Console.WriteLine("Cantidad inválida para retiro.");
        }
    }
}
```

```

        else if (cantidad > saldo)
        {
            Console.WriteLine("Fondos insuficientes. Saldo: $" + saldo);
        }
        else
        {
            saldo -= cantidad;
            Console.WriteLine("Retiro exitoso. Saldo: $" + saldo);
        }
    }

    public void ConsultarSaldo()
    {
        Console.WriteLine("Cuenta " + numeroCuenta + ", saldo: $" + saldo);
    }
}

class Program
{
    static void Main()
    {
        CuentaBancaria cuenta = new CuentaBancaria("12345-6");
        cuenta.Depositar(1000);    // saldo: 1000
        cuenta.Retirar(300);       // saldo: 700
        cuenta.Retirar(1000);      // falla: fondos insuficientes
        cuenta.Depositar(-50);     // falla: cantidad inválida
        cuenta.ConsultarSaldo();   // Cuenta 12345-6, saldo: $700
    }
}

```

Explicación: esta clase muestra el poder de la POO para hacer cumplir reglas del negocio. El saldo nunca se toca directamente desde Main: solo cambia a través de Depositar y Retirar, que verifican las reglas antes de actuar. Es imposible dejar la cuenta en negativo porque Retirar lo impide. El constructor garantiza que toda cuenta nazca con número y saldo cero. En Main probamos los casos felices y los casos de error: el retiro de 1000 con saldo de 700 se rechaza con un mensaje claro en lugar de romper el programa. Este patrón (métodos que validan antes de modificar el estado) es la base del encapsulamiento que profundizaremos en la Unidad 6.

Solución 6: inventario con arreglo de objetos. La combinación final: colección de objetos, cada uno con su propio estado.

```

using System;

class Producto

```

```

{
    string nombre;
    double precio;
    int stock;

    public Producto(string nombre, double precio, int stock)
    {
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }

    // Valor total de este producto: precio por unidades en stock.
    public double ValorEnInventario()
    {
        return precio * stock;
    }

    public void Mostrar()
    {
        Console.WriteLine(nombre + " - $" + precio + " (" + stock + " en stock)");
    }
}

class Program
{
    static void Main()
    {
        // El arreglo (contenedor) se crea con new...
        Producto[] inventario = new Producto[4];

        // ...pero cada objeto necesita SU propio new.
        inventario[0] = new Producto("Laptop", 25000, 5);
        inventario[1] = new Producto("Mouse", 350, 40);
        inventario[2] = new Producto("Teclado", 900, 25);
        inventario[3] = new Producto("Monitor", 4500, 12);

        double valorTotal = 0;
        Console.WriteLine("Inventario de la tienda:");
        foreach (Producto p in inventario)
        {
            p.Mostrar();
            valorTotal += p.ValorEnInventario();
        }
    }
}

```

```
        Console.WriteLine("Valor total del inventario: $" + valorTotal);  
    }  
}
```

Explicación: el arreglo inventario guarda referencias a 4 objetos Producto independientes. Cada posición se llena con su propio new: sin esos new, las casillas valdrían null y p.Mostrar() lanzaría NullReferenceException. El método ValorEnInventario encapsula el cálculo $\text{precio} * \text{stock}$ dentro del objeto: Main no necesita saber la fórmula, solo le pide a cada producto su valor y los suma (patrón acumular con objetos). El resultado: $25000*5 + 350*40 + 900*25 + 4500*12 = 125000 + 14000 + 22500 + 54000 = 215500$. Este programa es un mini sistema de inventarios real: con esta estructura base podrías agregar búsqueda de productos, descuentos o reportes.

Solución 7: tres en raya. Practica matrices de tipo char e impresión con formato.

```

using System;

class Program
{
    static void Main()
    {
        // Tablero de 3x3 con caracteres.
        char[,] tablero = new char[3, 3];

        // Inicializamos todo con espacios en blanco.
        for (int f = 0; f < tablero.GetLength(0); f++)
        {
            for (int c = 0; c < tablero.GetLength(1); c++)
            {
                tablero[f, c] = ' ';
            }
        }

        // Colocamos las jugadas: X al centro, O en la esquina.
        tablero[1, 1] = 'X';
        tablero[0, 0] = 'O';

        // Imprimimos con formato de gato.
        for (int f = 0; f < tablero.GetLength(0); f++)
        {
            // Cada fila: casilla | casilla | casilla
            Console.WriteLine(" " + tablero[f, 0] + " | " + tablero[f, 1] + " | " +
tablero[f, 2] + " ");
            // Línea separadora entre filas (pero no después de la última).
            if (f < tablero.GetLength(0) - 1)
            {
                Console.WriteLine("---+---+---");
            }
        }
    }
}

```

Explicación: los char se escriben con comillas simples ('X'), a diferencia de los string que usan comillas dobles. Inicializar la matriz con espacios es importante: sin eso, las casillas tendrían el valor por defecto de char, que es un carácter invisible raro, y el tablero se vería mal. La impresión combina los valores de la matriz con texto decorativo (" | " y las líneas "---+---+---"). El if dentro del ciclo de impresión evita dibujar una línea separadora después de la última fila: un detalle de presentación que distingue un programa cuidado. El tablero resultante se ve así:

O | | ---+---+--- | X | ---+---+--- | |

Solución 8: clase Rectangulo con validación. Constructor que garantiza dimensiones válidas y métodos que calculan a partir del estado.

```
using System;

class Rectangulo
{
    double baseRect;
    double altura;

    // Constructor con validación: dimensiones no positivas se ajustan a 1.
    public Rectangulo(double baseRect, double altura)
    {
        if (baseRect > 0)
        {
            this.baseRect = baseRect;
        }
        else
        {
            this.baseRect = 1;
        }

        if (altura > 0)
        {
            this.altura = altura;
        }
        else
        {
            this.altura = 1;
        }
    }

    // Constructor sin parámetros: rectángulo unitario de 1x1.
    public Rectangulo()
    {
        this.baseRect = 1;
        this.altura = 1;
    }

    public double CalcularArea()
    {
        return baseRect * altura;
    }
}
```

```

    public double CalcularPerimetro()
    {
        return 2 * (baseRect + altura);
    }

    public void Mostrar()
    {
        Console.WriteLine("Rectángulo de " + baseRect + " x " + altura +
            ": área = " + CalcularArea() +
            ", perímetro = " + CalcularPerimetro());
    }
}

class Program
{
    static void Main()
    {
        Rectangulo r1 = new Rectangulo(5, 3);    // normal: área 15, perímetro 16
        Rectangulo r2 = new Rectangulo(-4, 10); // base inválida -> se ajusta a 1
        Rectangulo r3 = new Rectangulo();        // 1x1

        r1.Mostrar();
        r2.Mostrar(); // Rectángulo de 1 x 10: área = 10, perímetro = 22
        r3.Mostrar(); // Rectángulo de 1 x 1: área = 1, perímetro = 4
    }
}

```

Explicación: el constructor valida cada dimensión por separado: si recibe un valor no positivo, lo sustituye por 1 en lugar de permitir un rectángulo imposible. Así ningún objeto Rectangulo nace en estado inválido. Los métodos CalcularArea y CalcularPerimetro no reciben parámetros: trabajan con los campos del objeto, que es la forma natural en POO. El caso r2 demuestra la validación funcionando: la base de -4 se convierte en 1 y el programa sigue sin romperse. El constructor sin parámetros ofrece una tercera forma de nacer (el rectángulo unitario), mostrando la sobrecarga de constructores en acción.

Resumen de la unidad

- Un arreglo guarda varios valores del mismo tipo bajo un solo nombre, en posiciones numeradas llamadas índices que empiezan en cero.
- Se declara con corchetes (int[]) y se crea con new indicando el tamaño; también se puede inicializar con valores directos entre llaves.

- La regla de oro: en un arreglo de tamaño N, los índices válidos van de 0 a N - 1; el último índice válido siempre es Length - 1.
- La propiedad Length devuelve el tamaño del arreglo; úsala en los ciclos en lugar de números mágicos.
- Recorre con for cuando necesites el índice o modificar elementos (condición: $i < \text{arreglo.Length}$) y con foreach cuando solo necesites leer valores.
- Los patrones fundamentales con arreglos son: acumular (sumar), buscar (encontrar un valor o el máximo/mínimo) y contar (cuántos cumplen una condición).
- Los arreglos se pasan a los métodos por referencia: el método trabaja sobre el original, no sobre una copia.
- Las matrices (`int[,]`) son arreglos de dos dimensiones con filas y columnas; se accede con dos índices: `matriz[fila, columna]`.
- Usa `GetLength(0)` para filas y `GetLength(1)` para columnas; `Length` en una matriz devuelve el total de elementos.
- Las matrices se recorren con dos ciclos anidados: el exterior para filas, el interior para columnas.
- Los arreglos dentados (`int[][]`) son arreglos de arreglos donde cada fila puede tener distinta longitud.
- Una clase es el plano (la definición) y un objeto es la casa construida (la instancia concreta); no los confundas.
- La palabra `new` es la "constructora": crea el objeto en memoria a partir de la clase.
- Los campos son los datos de cada objeto; los métodos de instancia son sus comportamientos y acceden directamente a esos datos.
- La sintaxis `objeto.miembro` (el punto) es como hablarle al objeto: `juan.Presentarse()` le pide a juan que se presente.
- Las clases son tipos de referencia: una variable guarda la dirección del objeto, no el objeto; dos variables pueden apuntar al mismo objeto.
- Un constructor es un método especial (mismo nombre que la clase, sin tipo de retorno) que se ejecuta al hacer `new` e inicializa el objeto.
- El constructor con parámetros exige los datos esenciales al nacer; el constructor por defecto (sin parámetros) lo regala C# solo si no defines ningún constructor.

- La palabra `this` se refiere al objeto actual y sirve para distinguir un campo de un parámetro con el mismo nombre.
- `IndexOutOfRangeException`: accediste a un índice que no existe; prevénlo con `i < Length` y validando índices que vengan del usuario.
- `NullReferenceException`: intentaste usar un objeto que es `null` (sin `new`); cada objeto necesita su `new` antes de usarse, incluyendo cada casilla de un arreglo de objetos.
- Buenas prácticas: usa `foreach` para lectura, `Length` en lugar de números mágicos, constructores que exijan datos esenciales, valida entradas, una responsabilidad por clase y nombres claros y descriptivos.

UNIDAD 6: Encapsulamiento, herencia y polimorfismo

En la unidad anterior diste el salto a la programación orientada a objetos: aprendiste qué son las clases y los objetos, cómo crearlos con `new` y cómo inicializarlos con constructores. Tus clases ya funcionan, pero hay un problema: tal como las dejamos, son como casas sin puertas ni candados. Cualquiera puede entrar y mover los muebles a su antojo. Si tu clase `CuentaBancaria` tiene el campo `saldo` accesible directamente, cualquier parte del programa puede hacer `cuenta.saldo = -1000000` y romper todas las reglas que tanto te costó programar en los métodos `Depositar` y `Retirar`. Los datos de tus objetos están expuestos, desprotegidos, a merced de quien quiera modificarlos. Eso, en un programa real, es una bomba de tiempo.

En esta unidad vas a aprender los tres pilares que convierten tus clases de juguete en clases profesionales: encapsulamiento, herencia y polimorfismo. El encapsulamiento te enseña a proteger los datos de tus objetos: a decidir qué se muestra al exterior y qué se mantiene en privado, usando los modificadores `public` y `private`, las propiedades con `get` y `set`, las auto-propiedades y la validación en los setters. Es como ponerle puertas con chapa a tu casa: tú decides quién entra y con qué reglas.

La herencia te permite crear nuevas clases a partir de clases existentes, heredando sus datos y comportamientos y agregando los propios. Es como una familia: los hijos heredan rasgos de los padres, pero cada hijo es único y tiene sus propias características. Con herencia modelarás jerarquías naturales del mundo real: un `Gerente` es un `Empleado` con responsabilidades extra, un `Perro` es un `Animal` con comportamientos propios. Aprenderás la palabra base para comunicarte con la clase padre, el modificador `protected` para compartir con los hijos sin exponer al mundo, y la analogía de la familia que te ayudará a diseñar jerarquías con sentido.

El polimorfismo, la joya de la corona, te permite tratar objetos diferentes de manera uniforme: pedirle a cualquier figura geométrica que calcule su área, sin importar si es un círculo, un cuadrado o un triángulo, y que cada una responda a su manera. Aprenderás las palabras `virtual` y `override`, las clases abstractas (plantillas incompletas que obligan a los hijos a completarlas) y `sealed` (el candado que impide más herencia). Y lo más importante: aprenderás cuándo usar cada mecanismo, porque un profesional no solo sabe las herramientas, sino cuándo conviene cada una.

También conocerás los errores clásicos de esta etapa: exponer campos públicos sin control, olvidar la palabra `virtual` y que el polimorfismo no funcione, abusar de la herencia para

reutilizar código cuando lo correcto era la composición. Y las buenas prácticas que separan al programador que "sabe POO" del que "domina POO": favorecer la composición sobre la herencia, diseñar jerarquías que tengan sentido en el mundo real, y validar siempre en los setters.

Al terminar esta unidad tus clases serán robustas y profesionales: datos protegidos con validación automática, jerarquías de clases que modelan relaciones reales, y código polimórfico que trabaja con familias enteras de objetos sin conocer sus detalles. Podrás leer y entender código orientado a objetos de nivel profesional, diseñar tus propias jerarquías para tus proyectos (tu tienda, tu sistema escolar, tu videojuego) y tendrás la base completa de la POO que te pedirán en cualquier entrevista de trabajo de programación. Los tres pilares de esta unidad aparecen, literalmente, en todas las entrevistas técnicas para puestos de C#. Domínalos y llevarás ventaja.

Explicación detallada

Clase 16: Encapsulamiento - ponle puertas a tus objetos

El encapsulamiento es el primero de los tres pilares de la POO y su idea es simple pero poderosa: esconder los detalles internos de un objeto y exponer solo lo necesario a través de una interfaz controlada. Piensa en tu teléfono celular: tú lo usas a través de la pantalla, los botones y las apps (la interfaz pública), pero no necesitas saber cómo funciona el procesador por dentro, ni puedes modificar directamente el voltaje de la batería (los detalles privados). Si pudieras cambiar el voltaje a mano, probablemente quemarías el teléfono. El fabricante encapsuló los componentes delicados y te dio controles seguros. Tus clases deben hacer lo mismo con sus datos.

En C#, el encapsulamiento se logra con modificadores de acceso, que son palabras clave que controlan quién puede ver y usar cada miembro de la clase. Los dos fundamentales son `public` y `private`. Un miembro `public` es visible para todo el mundo: cualquier código puede usarlo. Un miembro `private` solo es visible dentro de la propia clase: es un secreto interno. Por defecto, si no escribes ningún modificador, los miembros de una clase son `private`. La estrategia profesional es: campos `private` siempre, y acceso controlado a través de miembros `public`. Así ningún código externo puede dejar tus datos en un estado inválido.

Veamos el problema con un ejemplo concreto. Imagina esta clase sin encapsulamiento: `class Producto { public string nombre; public double precio; }`. Como los campos son `public`, cualquier código puede hacer: `Producto p = new Producto(); p.precio = -500;`. Precio negativo, sin que nadie lo impida. El objeto quedó en un estado absurdo y el error se descubrirá mucho

después, cuando algún cálculo dé un resultado imposible de rastrear. Con encapsulamiento, el campo es `private` y el precio solo se puede modificar a través de un método o propiedad que valide: si alguien intenta asignar `-500`, la validación lo rechaza. El objeto se defiende solo. Esa es la gran idea: un objeto bien encapsulado es imposible de dejar en estado inválido desde afuera.

Ahora, si los campos son `private`, ¿cómo lee y modifica el mundo exterior los datos del objeto? A través de propiedades. Una propiedad en C# es un miembro que se ve como un campo desde afuera (se usa con la sintaxis simple `objeto.Precio = 100`) pero por dentro ejecuta código: un método `get` para leer y un método `set` para escribir. Es lo mejor de dos mundos: la comodidad de un campo con el control de un método. La sintaxis es: `public double Precio { get { return precio; } set { if (value >= 0) precio = value; } }`. Analicemos: `public double Precio` declara la propiedad pública de tipo `double`. El bloque `get` se ejecuta cuando alguien lee la propiedad (`double x = p.Precio;`) y normalmente devuelve el campo privado. El bloque `set` se ejecuta cuando alguien asigna (`p.Precio = 100;`) y recibe el valor asignado en la variable especial `value`. Dentro del `set` puedes validar: si `value` es negativo, lo ignoras o lanzas una excepción. Desde afuera, `p.Precio = -500` parece una asignación normal, pero por dentro tu validación la detiene. Magia controlada.

Las propiedades tienen convención de nombres: se escriben en `PascalCase` (`Precio`, `Nombre`, `Saldo`) mientras que los campos privados suelen ir en `camelCase` (`precio`, `nombre`, `saldo`). Esta convención te ayuda a distinguirlos a simple vista. Cuando el campo y la propiedad se llaman casi igual (`precio` y `Precio`), C# los distingue por la mayúscula inicial: no hay conflicto.

Si una propiedad no necesita validación ni lógica especial, C# ofrece las auto-propiedades: `public string Nombre { get; set; }`. Con esta sola línea, el compilador crea automáticamente un campo privado oculto y los métodos `get` y `set` simples. Es la forma corta y elegante de declarar una propiedad simple. Úsala cuando el dato no requiera validación: te ahorra escribir el campo y los bloques `get/set` a mano. Pero ojo: en cuanto necesites validar, debes escribir la propiedad completa con su campo privado explícito. Una trampa común es empezar con auto-propiedad y luego necesitar validación: tendrás que convertirla a propiedad completa, lo cual es un cambio pequeño pero debes saber hacerlo.

Las propiedades también pueden ser de solo lectura o de solo escritura. Una propiedad con solo `get` (`public string Matricula { get; }`) permite leer pero no modificar desde afuera: perfecta para datos que se fijan al nacer y no deben cambiar, como una matrícula o un folio. En C# moderno puedes inicializarla en el constructor o en la declaración. Una propiedad con solo `set` es rara en la práctica (¿para qué guardar algo que nunca podrás leer?), pero existe. También puedes tener un `set` privado: `public double Saldo { get; private set; }`. Esto significa que todo el mundo puede leer el saldo, pero solo la propia clase puede modificarlo (típicamente a través

de métodos como Depositar y Retirar). Es un patrón muy usado en clases como CuentaBancaria: el saldo es público para lectura, privado para escritura, y los métodos controlan los cambios con reglas.

La validación en los setters merece atención especial porque es donde el encapsulamiento muestra su verdadero valor. Piensa en todas las reglas que un objeto debe cumplir: el precio no puede ser negativo, el nombre no puede estar vacío, la edad debe estar entre 0 y 150, el stock no puede ser negativo. Cada una de estas reglas vive en el set de su propiedad. Cuando la regla está en el setter, se aplica siempre, sin excepción, sin importar desde qué parte del programa se asigne el valor. No dependes de que cada programador "se acuerde de validar": la validación es automática e inevitable. Esto elimina una categoría entera de errores.

¿Qué hacer cuando la validación falla? Tienes dos opciones. La primera es ignorar silenciosamente el valor inválido (o ajustarlo): `if (value >= 0) precio = value;`. El objeto simplemente no cambia. Es simple pero puede confundir: el programador cree que asignó el valor y no se entera de que fue rechazado. La segunda opción es lanzar una excepción: `if (value < 0) throw new ArgumentException("El precio no puede ser negativo");`. Esto detiene el programa con un mensaje claro en el punto exacto del error. Es más ruidoso pero mucho más fácil de depurar. Como regla general: en código de aprendizaje y programas pequeños, ignorar o ajustar está bien; en código profesional y bibliotecas, lanzar excepciones con mensajes claros es lo correcto, porque un dato inválido silencioso es un bug esperando a explotar.

Un detalle importante: el encapsulamiento no es solo "poner private a todo". Es diseñar la interfaz pública de tu clase con intención: ¿qué necesita saber el mundo exterior de este objeto? ¿Qué operaciones necesita realizar? Todo lo demás va en privado. Una clase bien encapsulada es fácil de usar (pocos miembros públicos, claros) y difícil de usar mal (las reglas se hacen cumplir solas). Cuando diseñes una clase, primero piensa en su interfaz pública y después en sus detalles privados.

public, private y los demás modificadores: quién puede ver qué

Profundicemos en los modificadores de acceso porque controlar la visibilidad es el corazón del encapsulamiento. C# tiene varios niveles, pero en esta unidad nos concentramos en los tres que usarás todos los días. `public`: visible para todo el mundo, sin restricciones. Es lo que forma la interfaz pública de tu clase: los métodos y propiedades que otros programadores (o tú mismo desde otras clases) usarán. `private`: visible solo dentro de la clase donde se declara. Es el nivel más restrictivo y el que protege tus campos y métodos auxiliares internos. `protected`: lo veremos a fondo en la sección de herencia, pero adelanto: visible dentro de la clase y dentro de sus clases hijas. Es el punto medio entre `private` y `public`, diseñado para la herencia.

La regla de oro del diseño: haz privados todos los campos, y expón solo lo necesario como público a través de propiedades y métodos. Si dudas entre `public` y `private` para un miembro, elige `private`; siempre puedes volverlo público después si hace falta, pero un miembro público que luego quieres ocultar rompe el código de todos los que lo usaban. Este principio se llama "mínimo privilegio": cada miembro debe tener la visibilidad mínima necesaria para hacer su trabajo.

Los métodos auxiliares internos también van en `private`. Por ejemplo, si tu clase tiene un método `ValidarDatos()` que solo se usa dentro de la propia clase, decláralo `private`: nadie de afuera necesita saber que existe. Esto mantiene tu interfaz pública limpia y te da libertad para cambiar los detalles internos sin afectar a nadie. Imagina una clase con 20 métodos públicos donde 15 son detalles internos: sería una pesadilla de usar. El encapsulamiento también es cortesía con quien usa tu código.

Clase 17: Herencia - las familias de clases

El segundo pilar de la POO es la herencia, y su idea central es la reutilización a través de relaciones naturales: una clase nueva (la clase derivada o hija) puede heredar los campos y métodos de una clase existente (la clase base o padre), y luego agregar sus propios miembros o modificar los heredados. Piensa en una familia: los hijos heredan rasgos de los padres (el color de ojos, la estatura), pero cada hijo además tiene sus propias características únicas y su propia personalidad. La herencia en programación funciona igual: la clase hija recibe todo lo de la clase padre y le suma lo suyo.

Veamos un ejemplo del mundo real: en una empresa hay empleados. Todos los empleados tienen nombre, salario y un método `Trabajar()`. Pero hay tipos especiales: los gerentes además tienen un bono anual y pueden `AsignarTareas()`; los desarrolladores además conocen lenguajes de programación y pueden `EscribirCodigo()`. Sin herencia, tendrías que repetir nombre, salario y `Trabajar()` en cada clase: código duplicado en tres lugares. Con herencia, creas la clase base `Empleado` con lo común, y luego `Gerente` y `Desarrollador` heredan de `Empleado` y agregan solo lo suyo. Si mañana cambia la forma de calcular el salario, lo cambias en un solo lugar: la clase base. Esa es la magia de la herencia: escribes lo común una vez y lo reutilizas en todas las hijas.

En C#, la herencia se declara con dos puntos después del nombre de la clase: `class Gerente : Empleado { ... }`. Esta línea dice: "Gerente hereda de Empleado; Gerente es un tipo especial de Empleado". La relación es "es un": un Gerente ES UN Empleado, un Perro ES UN Animal. Esta frase es tu prueba de fuego para decidir si la herencia es correcta: si no puedes decir "la clase hija ES UNA clase padre" con naturalidad, probablemente la herencia no es la herramienta adecuada (más adelante veremos que la composición suele ser mejor en esos

casos). "Un Gerente es un Empleado" suena perfecto. "Un Motor es un Coche" suena mal (un motor es PARTE de un coche, no UN coche): ahí va composición, no herencia.

Cuando una clase hija hereda, obtiene automáticamente todos los miembros public y protected de la base: no necesita reescribirlos. Puede usarlos directamente como si fueran suyos. Además, la hija puede agregar nuevos campos y métodos propios. Y aquí viene algo hermoso: donde sea que el código espere un Empleado, puedes pasar un Gerente, porque un Gerente ES UN Empleado. Esto se llama sustitución y es la base del polimorfismo que veremos en la siguiente sección. Por ejemplo, un método void PagarSueldo(Empleado e) puede recibir un Gerente sin problema.

La palabra clave base es el puente de comunicación con la clase padre. Su uso más común es en los constructores: cuando creas un Gerente con `new Gerente("Ana", 50000, 10000)`, el constructor de Gerente debe inicializar también los datos heredados (nombre y salario) que pertenecen a Empleado. En lugar de duplicar esa lógica, el constructor de la hija llama al constructor de la base: `public Gerente(string nombre, double salario, double bono) : base(nombre, salario) { this.bono = bono; }`. La parte `: base(nombre, salario)` ejecuta primero el constructor de Empleado con esos argumentos, y después el constructor de Gerente inicializa lo propio (el bono). Es como una cadena de nacimientos: primero nace la parte "empleado" y luego se le agrega la parte "gerente". Si la clase base no tiene constructor sin parámetros y la hija no llama a base con los argumentos correctos, el código no compila: C# te obliga a inicializar la parte heredada.

El modificador `protected` es el nivel de acceso diseñado para la herencia: un miembro `protected` es visible dentro de la propia clase y dentro de sus clases hijas, pero no para el mundo exterior. Imagina que la clase Empleado tiene un campo `protected double salarioBase;`. La clase Gerente (hija) puede leer y modificar `salarioBase` directamente, pero el código de Main no puede tocarlo. Es el punto medio perfecto: compartes detalles con tus hijas sin exponerlos al público. Úsalo para aquellos miembros internos que las hijas necesitarán para implementar sus comportamientos, pero que no forman parte de la interfaz pública.

C# tiene una regla importante: solo se permite herencia simple, es decir, una clase puede heredar de UNA sola clase base. No puedes hacer `class X : A, B`. Esto es intencional: la herencia múltiple causa problemas famosos (como el "problema del diamante") que complican el lenguaje. Si necesitas combinar comportamientos de varias fuentes, C# ofrece las interfaces, que verás más adelante en tu formación. Por ahora, quédate con: una clase, un padre. Afortunadamente, un padre puede tener muchos hijos, y los hijos pueden ser padres de otros: las jerarquías pueden tener varios niveles (Empleado -> Gerente -> GerenteSenior), aunque en la práctica más de dos o tres niveles se vuelve difícil de manejar.

Hablemos de cuándo NO usar herencia, porque este es uno de los errores más caros. La herencia es para relaciones "es un" genuinas y estables. Si solo quieres reutilizar código sin que haya una relación real, la herencia te traerá problemas: acoplamiento rígido (cambios en la base rompen a las hijas), jerarquías artificiales difíciles de entender y rigidez ante cambios futuros. La alternativa es la composición: en lugar de heredar, la clase contiene un objeto de la otra clase como campo ("tiene un" en lugar de "es un"). Un Coche TIENE UN Motor (composición), no ES UN motor. Un Estudiante TIENE calificaciones (ya lo hiciste en la Unidad 5 con el arreglo), no ES calificaciones. La regla profesional: favorece la composición sobre la herencia; usa herencia solo cuando la relación "es un" sea clara, natural y estable en el tiempo.

Otro detalle: los miembros `private` de la base NO se heredan de forma accesible. La hija recibe la memoria de esos campos (existen en el objeto), pero no puede tocarlos directamente: solo a través de los miembros `public` o `protected` de la base. Esto es encapsulamiento funcionando a través de la herencia: ni siquiera tus hijas pueden manosear tus secretos privados. Si quieres que las hijas accedan, usa `protected`; si no, `private`.

La analogía de la familia en detalle

Vamos a expresar la analogía de la familia porque es la mejor herramienta mental para diseñar jerarquías. Piensa en la familia García. El abuelo (la clase base) definió el apellido, algunas tradiciones y la receta secreta de los tamales. Los hijos (clases derivadas) heredan el apellido y las tradiciones automáticamente: no tienen que "reinventar" ser García. Cada hijo además tiene su propia profesión, sus propios gustos: eso es lo que cada clase hija agrega. Los nietos heredan de sus padres (que ya traían lo del abuelo): las jerarquías acumulan.

Ahora, hay cosas del abuelo que son públicas para todo el pueblo (el apellido: `public`), cosas que solo se comparten en familia (la receta de los tamales: `protected`, los hijos la conocen pero los vecinos no) y cosas que el abuelo se llevó a la tumba (sus secretos personales: `private`, ni los hijos los conocen). Esta analogía mapea perfectamente a `public`, `protected` y `private`. Y como en las familias reales, cada hijo puede hacer las cosas a su manera: el abuelo contaba chistes de una forma, el hijo los cuenta a su manera (eso será el `override` en la sección de polimorfismo).

La analogía también advierte contra los excesos: una familia con demasiadas generaciones bajo el mismo techo se vuelve difícil de manejar. Igual con las jerarquías: dos o tres niveles están bien; si llegas a seis niveles de herencia, algo anda mal en tu diseño. Y así como no adoptas a alguien como hijo solo para que te ayude con la mudanza (relación artificial), no crees herencias artificiales solo para reutilizar un par de métodos: para eso está la composición.

Clase 18: Polimorfismo - muchas formas, una sola interfaz

El tercer pilar es el polimorfismo, una palabra griega que significa "muchas formas". Su idea: un mismo mensaje (una llamada a un método) puede producir comportamientos diferentes según el objeto que lo reciba. El ejemplo clásico: tienes figuras geométricas (círculo, cuadrado, triángulo) y a cada una le pides "calcula tu área". El mensaje es el mismo (CalcularArea()), pero cada figura responde a su manera: el círculo usa pi por radio al cuadrado, el cuadrado usa lado por lado, el triángulo usa base por altura entre dos. Tú, como programador, no necesitas saber qué figura es cada una: les hablas a todas igual y cada una hace lo correcto. Esa uniformidad con comportamientos diversos es el polimorfismo.

¿Por qué es tan poderoso? Porque te permite escribir código que trabaja con la clase base sin conocer las clases hijas, incluso hijas que se inventen en el futuro. Imagina un método void ImprimirArea(Figura f) { Console.WriteLine(f.CalcularArea()); }. Este método funciona con cualquier figura: las de hoy y las que programes el año que entra. Si mañana agregas la clase Pentagono, ImprimirArea la acepta sin cambiar ni una línea. Esto se llama código extensible: abierto a nuevos tipos, cerrado a modificaciones. Es una de las propiedades más valiosas en el software profesional, porque los programas reales viven años y cambian constantemente.

En C#, el polimorfismo se logra con dos palabras clave que trabajan en pareja: virtual y override. En la clase base, marcas el método como virtual: `public virtual double CalcularArea() { return 0; }`. La palabra virtual significa "este método puede ser redefinido por las clases hijas; esta es la versión por defecto". En cada clase hija, redefines el método con override: `public override double CalcularArea() { return Math.PI * radio * radio; }`. La palabra override significa "estoy reemplazando la versión de mi padre con mi propia versión". Cuando llamas al método a través de una referencia de la clase base, C# ejecuta automáticamente la versión override de la clase hija real del objeto. Esa selección automática en tiempo de ejecución se llama despacho dinámico y es el corazón del polimorfismo.

Aquí hay un detalle crucial que causa el error más común de esta sección: si olvidas la palabra virtual en la base (o el override en la hija), el polimorfismo NO funciona. Sin virtual, cuando llamas al método desde una referencia de la base, se ejecuta la versión de la base aunque el objeto real sea de la hija. C# te avisará con una advertencia si defines un método con el mismo nombre sin override (te sugiere usar new, que es otra cosa: ocultamiento, no polimorfismo). La regla es simple y absoluta: para polimorfismo, virtual en la base y override en la hija, siempre en pareja. Si falta una de las dos, no hay polimorfismo real.

Veamos el mecanismo completo con la jerarquía de figuras. La clase base Figura define `public virtual double CalcularArea()` con una implementación por defecto (quizá devuelve 0 o lanza una excepción). Las hijas Circulo, Cuadrado y Triangulo hacen override con sus fórmulas.

Ahora el código cliente: `Figura[] figuras = new Figura[3]; figuras[0] = new Circulo(5); figuras[1] = new Cuadrado(4); figuras[2] = new Triangulo(6, 3);`. Fíjate: el arreglo es de tipo `Figura` (la base), pero cada casilla guarda un objeto de una hija distinta. Esto es válido por la sustitución: un `Circulo` ES UNA `Figura`. Luego: `foreach (Figura f in figuras) { Console.WriteLine(f.CalcularArea()); }`. La variable `f` es de tipo `Figura`, pero en cada vuelta C# ejecuta la versión `override` correspondiente al objeto real: 78.5, 16, 9. Un solo ciclo, un solo mensaje, tres comportamientos distintos. Eso es polimorfismo en acción.

Las clases abstractas llevan esta idea un paso más allá. Una clase abstracta es una clase incompleta a propósito: no se pueden crear objetos directamente de ella (no puedes hacer `new Figura()` si `Figura` es abstracta), y puede contener métodos abstractos, que son métodos sin implementación: `public abstract double CalcularArea();`. Fíjate: no tiene cuerpo, termina en punto y coma. Un método abstracto es una promesa y una obligación: la clase base promete que "toda figura sabe calcular su área", y obliga a cada hija concreta a cumplirla con su propio `override`. Si una hija no implementa todos los métodos abstractos, ella misma debe declararse abstracta. ¿Cuándo usar clases abstractas? Cuando la clase base es un concepto tan general que no tiene sentido crear objetos de ella ni darle una implementación por defecto. ¿Qué área tendría "una figura en general"? Ninguna: el concepto solo cobra sentido en las hijas concretas. En ese caso, `abstract` es lo correcto: expresa tu intención con claridad y el compilador te ayuda obligando a las hijas a implementar.

La palabra `sealed` es el candado: una clase `sealed` no puede tener hijas (`class CajaFuerte sealed` o `public sealed class Utilidades`), y un método `override` sellado (`sealed override`) no puede ser redefinido por las hijas de la hija. ¿Para qué sirve? Para proteger diseños que no deben extenderse: clases de seguridad, implementaciones que dependen de detalles internos frágiles, o simplemente para expresar "este diseño está terminado, no lo heredes". También tiene un pequeño beneficio de rendimiento (el compilador puede optimizar llamadas a métodos sellados), pero su valor principal es de diseño: comunica intención.

¿Cuándo usar cada mecanismo? Esta es la pregunta de profesional. Usa una clase normal con métodos `virtual` cuando la base tenga una implementación por defecto razonable y las hijas solo necesiten ajustar algunos comportamientos. Ejemplo: la clase `Empleado` con método `virtual CalcularBono()` que devuelve un bono base, y `Gerente` hace `override` para agregar su extra. Usa una clase abstracta cuando la base sea un concepto general sin implementación sensata por defecto, y quieras obligar a las hijas a implementar. Ejemplo: `Figura` con `CalcularArea()` abstracto. Usa `sealed` cuando quieras impedir más herencia por seguridad o diseño. Y recuerda la alternativa silenciosa: si no hay relación "es un", no uses ninguno de estos: usa composición.

El polimorfismo también funciona con propiedades, no solo con métodos: puedes declarar `public virtual string Descripcion { get; ... }` en la base y hacer `override` en las hijas. Y los métodos abstractos se implementan en las hijas con `override` igualmente. Todo el sistema es coherente: `virtual/abstract` en la base, `override` en la hija.

Errores comunes y cómo evitarlos

Los errores de esta unidad son más sutiles que los de la anterior, porque el código compila y "casi" funciona. Presta atención extra.

Error uno: exponer campos públicos en lugar de propiedades. Es el pecado original del encapsulamiento: `public double precio;` permite asignar cualquier valor sin validación. Tal vez hoy no necesites validar, pero mañana sí, y cambiar un campo público a propiedad rompe la compatibilidad en algunos escenarios. La buena práctica profesional es usar propiedades (aunque sea auto-propiedades) desde el principio: `public double Precio { get; set; }`. Te cuesta lo mismo escribirlo y te deja la puerta abierta para agregar validación después sin romper nada.

Error dos: olvidar `virtual` en la base o `override` en la hija. El síntoma: llamas al método desde una referencia de la clase base y siempre se ejecuta la versión de la base, ignorando la de la hija. El programa compila (a lo sumo con una advertencia) pero el comportamiento es incorrecto. Diagnóstico: revisa que el método base diga `virtual` (o `abstract`) y el de la hija diga `override`, con la misma firma (mismo nombre, mismos parámetros, mismo tipo de retorno). Si C# te muestra una advertencia sugiriendo `new`, es señal de que olvidaste `override`: no uses `new` para "arreglarlo", porque `new` oculta el método en lugar de redefinirlo polimórficamente y el comportamiento seguirá siendo el de la base cuando llames desde una referencia base.

Error tres: mal uso de la herencia para reutilizar código. El síntoma típico: jerarquías que suenan raras al leerlas ("un Avion es un Vehiculo" está bien, pero "un Pato es un... ¿qué? ¿Animal que vuela? ¿que nada?"). O clases hijas que heredan métodos que no usan o que tienen que desactivar. Si tu hija necesita "desheredar" comportamientos del padre, la jerarquía está mal diseñada. La solución: prefiere composición. En lugar de `class Pato : AnimalVolador`, considera que Pato tenga los comportamientos que necesita por composición. Pregúntate siempre: ¿la relación "es un" es clara, natural y estable? Si dudas, composición.

Error cuatro: no llamar a base en el constructor de la hija cuando la base no tiene constructor sin parámetros. El compilador te lo impide con un error, así que no es silencioso, pero confunde: "¿por qué me exige llamar al constructor de mi padre?". Porque la parte heredada del objeto debe nacer correctamente antes de agregar lo propio. La solución: agrega `base(argumentos)` al constructor de la hija con los valores que la base necesita.

Error cinco: intentar crear objetos de una clase abstracta. `new Figura()` donde `Figura` es abstracta no compila. Es intencional: la clase abstracta es un concepto incompleto. Si necesitas un objeto, crea una de las hijas concretas. Este error suele indicar que estás pensando en la base como algo concreto cuando deberías pensar en las hijas.

Error seis: validación solo en el constructor pero no en el setter (o viceversa). Si validas el precio en el constructor pero la propiedad `Precio` permite asignar negativos después, tu validación inicial no sirve de nada: el objeto puede quedar inválido en cualquier momento. La regla: la validación vive en el setter (o en un método privado de validación llamado desde ambos), de modo que TODO cambio de valor pase por la regla, venga del constructor o de una asignación posterior. Un truco elegante: que el constructor asigne a través de la propiedad (`this.Precio = precio;`) en lugar de directo al campo, así la validación del setter se aplica también al nacer.

Buenas prácticas profesionales

Primero: favorece la composición sobre la herencia. Es el consejo más repetido en la industria por una razón: la composición (una clase contiene objetos de otras) es más flexible que la herencia (una clase extiende a otra). Con composición puedes cambiar comportamientos en tiempo de ejecución, combinar piezas libremente y evitar jerarquías rígidas. Usa herencia para relaciones "es un" genuinas y estables; para todo lo demás, composición. Cuando dudes, elige composición.

Segundo: diseña jerarquías con sentido en el mundo real. Cada nivel de la jerarquía debe representar un concepto que existiría aunque no hubiera computadoras: Empleado, Gerente, Figura, Circulo. Si tu jerarquía solo tiene sentido para el programa y no para el dominio del problema, probablemente estás forzando la herencia. Nombra las clases base con conceptos generales reales y pregúntate en cada hija: "¿esto ES UN ejemplo de la base?".

Tercero: mantén las jerarquías poco profundas. Dos o tres niveles son ideales; más de cuatro es una señal de alerta. Las jerarquías profundas son difíciles de entender (¿qué hace realmente este método después de 5 overrides?), frágiles (un cambio arriba rompe todo abajo) y difíciles de probar. Si tu jerarquía crece demasiado, considera dividirla o usar composición.

Cuarto: programa contra la base, no contra las hijas. Cuando escribas métodos que trabajen con jerarquías, recibe parámetros del tipo de la clase base (`void ImprimirArea(Figura f)`) y deja que el polimorfismo haga su trabajo. Así tu código funciona con hijas que aún no existen. Evita preguntar "¿de qué tipo eres?" con `if` o `switch` sobre tipos: cada vez que agregues una hija tendrías que actualizar esos condicionales, mientras que el polimorfismo lo resuelve solo.

Quinto: sella lo que no deba heredarse y haz abstracto lo incompleto. Si una clase representa un concepto completo que no debe extenderse, márcala sealed y dilo con claridad. Si una clase base no tiene sentido por sí sola, hazla abstracta en lugar de darle implementaciones ficticias por defecto. Estas palabras clave son documentación ejecutable: le dicen al compilador y a otros programadores cuáles eran tus intenciones de diseño.

Sexto: valida en los setters, no solo en los constructores, y haz que los constructores usen las propiedades. Así garantizas que el objeto sea válido al nacer y durante toda su vida. Un objeto que se defiende solo es un objeto en el que puedes confiar.

Ejemplos paso a paso

Ejemplo 1: Encapsulamiento con propiedades y validación - clase Producto

Vamos a reconstruir la clase Producto de la unidad anterior, pero ahora como un profesional: campos privados, propiedades con validación en los setters, auto-propiedades donde no se necesita validación, y constructores que usan las propiedades para validar al nacer.

```
using System;

class Producto
{
    // CAMPOS PRIVADOS: nadie de afuera puede tocarlos directamente.
    // Por convención van en camelCase.
    string nombre;
    double precio;
    int stock;

    // PROPIEDAD con validación: el nombre no puede estar vacío.
    // PascalCase por convención.
    public string Nombre
    {
        get { return nombre; }
        set
        {
            // value es el valor que alguien intenta asignar.
            if (value != null && value != "")
            {
                nombre = value;
            }
            else
            {

```



```

        Console.WriteLine("El nombre no puede estar vacío.");
    }
}

// PROPIEDAD con validación: el precio no puede ser negativo.
public double Precio
{
    get { return precio; }
    set
    {
        if (value >= 0)
        {
            precio = value;
        }
        else
        {
            Console.WriteLine("El precio no puede ser negativo. No se cambió.");
        }
    }
}

// PROPIEDAD con validación: el stock no puede ser negativo.
public int Stock
{
    get { return stock; }
    set
    {
        if (value >= 0)
        {
            stock = value;
        }
        else
        {
            Console.WriteLine("El stock no puede ser negativo. No se cambió.");
        }
    }
}

// AUTO-PROPIEDAD: la categoría no necesita validación,
// así que usamos la forma corta. El compilador crea
// el campo privado oculto por nosotros.
public string Categoria { get; set; }

// PROPIEDAD DE SOLO LECTURA: se calcula, no se almacena.

```

```

// No tiene set: nadie puede asignarle un valor.
public double ValorEnInventario
{
    get { return precio * stock; }
}

// CONSTRUCTOR: asigna a través de las PROPIEDADES (no directo
// a los campos) para que la validación se aplique al nacer.
public Producto(string nombre, double precio, int stock, string categoria)
{
    Nombre = nombre;          // pasa por el set con validación
    Precio = precio;          // pasa por el set con validación
    Stock = stock;            // pasa por el set con validación
    Categoria = categoria;    // auto-propiedad, asignación directa
}

public void MostrarFicha()
{
    Console.WriteLine(Nombre + " [" + Categoria + "]");
    Console.WriteLine("Precio: $" + Precio + " | Stock: " + Stock);
    Console.WriteLine("Valor en inventario: $" + ValorEnInventario);
}
}

class Program
{
    static void Main()
    {
        Producto laptop = new Producto("Laptop Gamer", 25000, 5, "Electrónica");
        laptop.MostrarFicha();

        Console.WriteLine("\nIntentando romper las reglas:");
        laptop.Precio = -100;    // rechazado por el set: el precio no cambia
        laptop.Stock = -20;      // rechazado por el set: el stock no cambia
        laptop.Nombre = "";      // rechazado por el set: el nombre no cambia

        Console.WriteLine("\nDespués de los intentos inválidos:");
        laptop.MostrarFicha(); // todo sigue igual: el objeto se defendió solo

        Console.WriteLine("\nAsignaciones válidas sí funcionan:");
        laptop.Precio = 22000;   // aceptado
        laptop.MostrarFicha();
    }
}

```

Explicación línea por línea: los tres campos (nombre, precio, stock) son privados: ningún código fuera de la clase puede leerlos ni modificarlos directamente. Cada uno tiene su propiedad pública con el mismo nombre en PascalCase. Dentro de cada set, la variable especial `value` contiene el valor que se intenta asignar, y solo se guarda en el campo si cumple la regla; si no, se muestra un mensaje y el campo conserva su valor anterior. La auto-propiedad `Categoria` demuestra la forma corta cuando no hay validación: una sola línea reemplaza `campo + get + set`. La propiedad `ValorEnInventario` solo tiene `get`: calcula `precio * stock` al momento de leerla, sin almacenar nada; es un dato derivado que siempre está actualizado.

El constructor es la parte elegante: en lugar de asignar a los campos directamente (`this.precio = precio`), asigna a las propiedades (`Precio = precio`). Como la asignación a una propiedad ejecuta su `set`, la validación se aplica automáticamente al nacer el objeto. Si alguien crea `new Producto("X", -50, 3, "Y")`, el precio negativo se rechaza y queda en 0 (el valor por defecto del campo). En `Main` demostramos el encapsulamiento en acción: los tres intentos de asignar valores inválidos son rechazados y el objeto conserva su estado válido. El objeto se defiende solo, sin que `Main` tenga que validar nada. Esa es la promesa del encapsulamiento: reglas que se cumplen siempre, en todas partes, automáticamente.

Ejemplo 2: Herencia - jerarquía Empleado y Gerente

Ahora construimos la jerarquía clásica: la clase base `Empleado` con lo común (nombre, salario, método `Trabajar`) y la clase derivada `Gerente` que hereda todo y agrega bono y el método `AsignarTarea`. Veremos la sintaxis de herencia, el constructor con base y el modificador `protected`.

```
using System;

// CLASE BASE: lo común a todos los empleados.
class Empleado
{
    // protected: las clases hijas sí pueden verlo, el mundo exterior no.
    protected string nombre;
    protected double salario;

    // El constructor de la base inicializa lo común.
    public Empleado(string nombre, double salario)
    {
        this.nombre = nombre;
        this.salario = salario;
    }
}
```

```

// Propiedad pública de solo lectura: el mundo puede ver el nombre,
// pero no cambiarlo directamente.
public string Nombre
{
    get { return nombre; }
}

// Método común: todo empleado sabe trabajar.
// virtual: las hijas podrán redefinirlo si lo necesitan.
public virtual void Trabajar()
{
    Console.WriteLine(nombre + " está trabajando.");
}

// Método común: todo empleado cobra su salario.
public double CalcularSalarioMensual()
{
    return salario / 12;
}

public void MostrarInfo()
{
    Console.WriteLine("Empleado: " + nombre + ", salario anual: $" + salario);
}
}

// CLASE DERIVADA: Gerente ES UN Empleado con extras.
// Los dos puntos indican la herencia.
class Gerente : Empleado
{
    // Campo propio: solo los gerentes tienen bono.
    double bonoAnual;

    // El constructor de la hija llama al de la base con : base(...).
    // Primero se ejecuta Empleado(nombre, salario), luego lo propio.
    public Gerente(string nombre, double salario, double bonoAnual)
        : base(nombre, salario)
    {
        this.bonoAnual = bonoAnual;
    }

    // Propiedad con validación para el bono.
    public double BonoAnual
    {
        get { return bonoAnual; }
    }
}

```

```

        set
        {
            if (value >= 0)
            {
                bonoAnual = value;
            }
        }
    }

    // Método propio: solo los gerentes asignan tareas.
    public void AsignarTarea(string tarea)
    {
        Console.WriteLine(nombre + " asignó la tarea: " + tarea);
    }

    // Override: el gerente trabaja a su manera (dirigiendo).
    // Usa los campos protected heredados directamente.
    public override void Trabajar()
    {
        Console.WriteLine(nombre + " está dirigiendo al equipo.");
    }

    // El gerente muestra su info extendida: reutiliza la base
    // y agrega lo suyo. base.MostrarInfo() llama a la versión del padre.
    public void MostrarInfoGerente()
    {
        base.MostrarInfo();
        Console.WriteLine("Bono anual: $" + bonoAnual);
    }
}

class Program
{
    static void Main()
    {
        // Un empleado común usa lo de la base.
        Empleado ana = new Empleado("Ana", 360000);
        ana.Trabajar();          // Ana está trabajando.
        ana.MostrarInfo();

        Console.WriteLine();

        // Un gerente: nace con : base(...) inicializando lo heredado.
        Gerente luis = new Gerente("Luis", 600000, 120000);
        luis.Trabajar();         // Luis está dirigiendo al equipo. (override)
    }
}

```

```

        luis.AsignarTarea("Terminar el reporte"); // método propio
        luis.MostrarInfoGerente();

        Console.WriteLine();

        // SUSTITUCIÓN: donde se espera un Empleado, va un Gerente.
        // El método recibe Empleado, pero le pasamos un Gerente sin problema.
        PagarSueldo(ana); // empleado común
        PagarSueldo(luis); // gerente: es un empleado, así que entra
    }

    // Este método solo conoce la clase base Empleado.
    // Funciona con empleados y con cualquier hija futura.
    static void PagarSueldo(Empleado e)
    {
        Console.WriteLine("Pagando $" + e.CalcularSalarioMensual() + " a " +
e.Nombre);
    }
}

```

Explicación: la clase Empleado define lo común con campos protected (visibles para las hijas, invisibles para el mundo), un constructor que los inicializa, una propiedad de solo lectura para el nombre, y métodos comunes. El método Trabajar es virtual, anticipando que las hijas quieran redefinirlo. La clase Gerente hereda con : Empleado y automáticamente puede usar nombre, salario, Trabajar, CalcularSalarioMensual y MostrarInfo como si fueran suyos. Su constructor usa : base(nombre, salario) para que la base inicialice su parte antes de que la hija agregue el bono: el orden de nacimiento es base primero, hija después.

Gerente agrega su campo bonoAnual con propiedad validada, su método propio AsignarTarea (que Empleado no tiene), y hace override de Trabajar para darle su propia versión. Fíjate que dentro de los métodos de Gerente usamos nombre directamente: es el campo protected heredado, accesible por ser hija. En Main, la sustitución brilla en PagarSueldo: el método declara que recibe un Empleado, pero acepta un Gerente porque todo Gerente ES UN Empleado. Este método no sabe qué es un Gerente ni le importa: trabaja con la base y funciona con toda la familia. Cuando agregues la clase Desarrollador mañana, PagarSueldo la aceptará sin cambios.

Ejemplo 3: Polimorfismo - figuras geométricas con área polimórfica

El ejemplo estrella del polimorfismo: una clase abstracta Figura con un método abstracto CalcularArea, tres hijas concretas (Circulo, Cuadrado, Triangulo) que lo implementan, y un programa que las trata a todas igual a través de la base.

```

using System;

// CLASE ABSTRACTA: el concepto "figura" es demasiado general
// para crear objetos de él. Nadie puede hacer new Figura().
// Obliga a las hijas a implementar CalcularArea.
abstract class Figura
{
    // Propiedad común con auto-propiedad.
    public string Color { get; set; }

    // MÉTODO ABSTRACTO: sin cuerpo, termina en punto y coma.
    // Es una promesa: "toda figura sabe calcular su área",
    // y una obligación: cada hija DEBE implementarlo con override.
    public abstract double CalcularArea();

    // Método concreto: las hijas lo heredan tal cual.
    public void MostrarColor()
    {
        Console.WriteLine("Color: " + Color);
    }
}

class Circulo : Figura
{
    double radio;

    public Circulo(double radio, string color)
    {
        this.radio = radio;
        this.Color = color; // propiedad heredada de la base
    }

    // Cumplimos la obligación: implementamos el método abstracto.
    public override double CalcularArea()
    {
        return Math.PI * radio * radio;
    }
}

class Cuadrado : Figura
{
    double lado;

    public Cuadrado(double lado, string color)
    {

```

```

        this.lado = lado;
        this.Color = color;
    }

    public override double CalcularArea()
    {
        return lado * lado;
    }
}

class Triangulo : Figura
{
    double baseTri;
    double altura;

    public Triangulo(double baseTri, double altura, string color)
    {
        this.baseTri = baseTri;
        this.altura = altura;
        this.Color = color;
    }

    public override double CalcularArea()
    {
        return (baseTri * altura) / 2;
    }
}

class Program
{
    static void Main()
    {
        // Arreglo de la CLASE BASE que guarda objetos de las hijas.
        // Cada casilla tiene un tipo distinto, pero todas SON Figuras.
        Figura[] figuras = new Figura[3];
        figuras[0] = new Circulo(5, "rojo");
        figuras[1] = new Cuadrado(4, "azul");
        figuras[2] = new Triangulo(6, 3, "verde");

        // POLIMORFISMO: un solo mensaje, tres comportamientos.
        // f es de tipo Figura, pero en cada vuelta se ejecuta
        // el override del objeto real (despacho dinámico).
        foreach (Figura f in figuras)
        {
            f.MostrarColor(); // método heredado, igual para todas
        }
    }
}

```



```

        Console.WriteLine("Área: " + f.CalcularArea()); // ¡polimórfico!
        Console.WriteLine();
    }

    // El método cliente solo conoce la base: funciona con
    // cualquier figura, incluso las que inventes en el futuro.
    ImprimirArea(new Circulo(10, "amarillo"));
    ImprimirArea(new Cuadrado(7, "negro"));
}

static void ImprimirArea(Figura f)
{
    Console.WriteLine("El área es: " + f.CalcularArea());
}
}

```

Explicación: Figura es abstracta porque "una figura en general" no tiene área calculable: el concepto solo existe en las hijas concretas. El método abstracto `CalcularArea` no tiene cuerpo: es un contrato. Cada hija lo implementa con `override` y su propia fórmula: el círculo usa $\text{Math.PI} * \text{radio} * \text{radio}$, el cuadrado $\text{lado} * \text{lado}$, el triángulo $(\text{base} * \text{altura}) / 2$. La propiedad `Color` y el método `MostrarColor` son miembros concretos heredados por todas: lo común se escribe una vez.

En Main, el arreglo `Figura[]` guarda tres tipos distintos gracias a la sustitución. El `foreach` es el momento del polimorfismo: la variable `f` es de tipo `Figura`, pero `f.CalcularArea()` ejecuta la versión `override` del objeto real en cada vuelta. El programa imprime 78.54 (círculo), 16 (cuadrado) y 9 (triángulo) con el mismo código. El método `ImprimirArea` demuestra la extensibilidad: recibe `Figura` y funciona con cualquier hija, presente o futura. Si mañana creas la clase `Pentagono : Figura` con su `override`, `ImprimirArea` la aceptará sin modificar ni una línea. Ese es el poder que las empresas pagan: código que crece sin romperse.

Ejemplo 4: sealed en acción y jerarquía de tres niveles

Un ejemplo más corto que muestra una jerarquía de tres niveles (`Persona` -> `Empleado` -> `GerenteSenior`) y el uso de `sealed` para cerrar un diseño. También veremos un método sellado.

```

using System;

class Persona
{
    public string Nombre { get; set; }
}

```

```

    public Persona(string nombre)
    {
        Nombre = nombre;
    }

    public virtual void Presentarse()
    {
        Console.WriteLine("Hola, soy " + Nombre);
    }
}

class Empleado : Persona
{
    public double Salario { get; set; }

    public Empleado(string nombre, double salario)
        : base(nombre)
    {
        Salario = salario;
    }

    public override void Presentarse()
    {
        Console.WriteLine("Hola, soy " + Nombre + ", empleado con salario $" +
Salario);
    }
}

// sealed: nadie puede heredar de GerenteSenior. El diseño termina aquí.
sealed class GerenteSenior : Empleado
{
    public int Nivel { get; set; }

    public GerenteSenior(string nombre, double salario, int nivel)
        : base(nombre, salario)
    {
        Nivel = nivel;
    }

    // sealed override: esta versión de Presentarse ya no puede
    // ser redefinida por nadie más (aunque la clase ya es sealed,
    // esto muestra la sintaxis para casos intermedios).
    public sealed override void Presentarse()
    {
        Console.WriteLine("Hola, soy " + Nombre + ", gerente senior nivel " + Nivel);
    }
}

```

```

    }
}

class Program
{
    static void Main()
    {
        // La cadena de constructores: Persona -> Empleado -> GerenteSenior.
        // Cada : base(...) inicializa su nivel antes de seguir bajando.
        GerenteSenior g = new GerenteSenior("Carmen", 900000, 3);
        g.Presentarse();

        // Polimorfismo a través de tres niveles: la referencia es
        // del abuelo Persona, pero se ejecuta la versión del nieto.
        Persona p = new GerenteSenior("Diego", 950000, 4);
        p.Presentarse(); // "Hola, soy Diego, gerente senior nivel 4"

        // Esto NO compila: no se puede heredar de una clase sealed.
        // class SubGerente : GerenteSenior { } // ERROR
    }
}

```

Explicación: la jerarquía tiene tres niveles: Persona (abuelo) -> Empleado (padre) -> GerenteSenior (hijo). Cada constructor llama a : base(...) pasando hacia arriba lo que el nivel superior necesita: GerenteSenior le pasa nombre y salario a Empleado, que le pasa el nombre a Persona. Es una cadena de inicialización de arriba hacia abajo en la declaración, pero de ejecución de la base hacia la hija: primero nace la parte Persona, luego la parte Empleado, luego la parte GerenteSenior. El método Presentarse se redefine en cada nivel con override, mostrando cómo el comportamiento se especializa al bajar.

La clase GerenteSenior es sealed: el diseño declara "aquí termina la jerarquía, nadie hereda de mí". Si alguien intenta `class SubGerente : GerenteSenior`, el compilador lo rechaza. El método Presentarse es sealed override: incluso si la clase no fuera sealed, este método ya no podría redefinirse más abajo. En Main, la línea `Persona p = new GerenteSenior(...)`; muestra el polimorfismo atravesando tres niveles: la referencia es del tipo abuelo, el objeto es del tipo nieto, y se ejecuta la versión del nieto. Tres niveles es un buen máximo práctico: más profundo y la jerarquía se vuelve difícil de seguir.

Práctica

Resuelve cada ejercicio por tu cuenta antes de mirar la solución. Los ejercicios de esta unidad te piden diseñar clases completas: piensa primero en la interfaz pública (¿qué necesita el mundo exterior?) y después en los detalles privados. Escribe el código, ejecútalo y experimenta.

1. Clase Estudiante encapsulada. Toma la clase Estudiante de la Unidad 5 y mejórala con encapsulamiento profesional: haz privados los campos nombre y matrícula, expónlos con propiedades (el nombre no puede estar vacío, la matrícula debe tener al menos 3 caracteres), agrega una auto-propiedad Carrera, y una propiedad de solo lectura Promedio que calcule el promedio de las calificaciones. El constructor debe asignar a través de las propiedades para validar al nacer.
2. Termómetro con validación. Crea una clase Termometro con una propiedad TemperaturaCelsius que solo acepte valores entre -273.15 (cero absoluto, nada puede estar más frío) y 1000. Si alguien intenta asignar un valor fuera de rango, recházalo con un mensaje. Agrega una propiedad de solo lectura TemperaturaFahrenheit que convierta automáticamente ($F = C * 9/5 + 32$). En Main prueba asignar valores válidos e inválidos y verifica que la conversión funcione.
3. Jerarquía de animales. Crea una clase base Animal con la propiedad Nombre y un método virtual HacerSonido() que imprima un sonido genérico. Crea dos hijas: Perro (que hace override con "¡Guau!") y Gato (override con "¡Miau!"), cada una con un método propio (el Perro tiene MoverCola(), el Gato tiene Ronronear()). En Main crea un arreglo Animal[] con un perro y un gato, recórrelo con foreach llamando a HacerSonido() polimórficamente, y demuestra la sustitución pasando ambos a un método que recibe Animal.
4. Figuras con perímetro. Extiende la jerarquía de figuras del Ejemplo 3: agrega a la clase abstracta Figura un segundo método abstracto CalcularPerimetro(). Implementa el override en Circulo ($2 * \pi * \text{radio}$), Cuadrado ($4 * \text{lado}$) y Triangulo (asume triángulo equilátero: $3 * \text{lado}$, agrega el campo que necesites). En Main recorre un arreglo de figuras imprimiendo área y perímetro de cada una con un solo ciclo.
5. Empleados con bono polimórfico. Crea una clase base Empleado con Nombre, SalarioBase y un método virtual CalcularPagoMensual() que devuelva SalarioBase / 12. Crea dos hijas: Vendedor (agrega ComisiónMensual y su override suma la comisión al pago) y Gerente (agrega BonoMensual y su override suma el bono). En Main crea una lista (arreglo) de Empleado con dos vendedores y un gerente, y calcula la nómina total del mes recorriendo el arreglo polimórficamente.

6. Clase sellada de utilidades. Crea una clase sealed llamada CalculadoraSegura con métodos para sumar, restar, multiplicar y dividir dos doubles. El método Dividir debe validar que el divisor no sea cero: si lo es, debe lanzar una excepción con un mensaje claro (usa `throw new ArgumentException("...")`). En Main prueba las cuatro operaciones incluyendo una división entre cero capturada con try/catch. Explica en un comentario por qué tiene sentido que esta clase sea sealed.
7. Reto: composición vs herencia. Diseña dos versiones de un sistema de motor y coche para comparar: primero, una versión con composición donde la clase Coche TIENE UN Motor (campo privado de tipo Motor con métodos Encender() y Apagar(), y el Coche expone Arrancar() que usa su motor). Segundo, escribe en comentarios por qué la herencia (class Coche : Motor) sería incorrecta aquí aplicando la prueba del "es un". En Main crea un coche, arráncalo y apágalo.
8. Reto: jerarquía de cuentas bancarias. Crea una clase abstracta CuentaBancaria con el número de cuenta, el saldo protegido (protected), un método abstracto CalcularInteresMensual() y un método concreto Depositar() con validación. Crea dos hijas: CuentaAhorros (interés del 0.5% mensual sobre el saldo) y CuentaCredito (interés del 2% mensual sobre el saldo deudor; su saldo puede ser negativo hasta un límite de -10000 validado en el setter). En Main crea una de cada una, deposita, calcula intereses y muestra cómo el polimorfismo permite tratarlas igual.

Soluciones

Solución 1: Estudiante encapsulada. La mejora profesional de la clase de la unidad anterior.

```
using System;

class Estudiante
{
    // Campos privados: el mundo exterior no los toca.
    string nombre;
    string matricula;
    int[] calificaciones = new int[5];
    int totalCalifs = 0;

    // Propiedad con validación: nombre no vacío.
    public string Nombre
    {
        get { return nombre; }
        set
        {
            if (value != null && value.Trim() != "")
```

```

        {
            nombre = value;
        }
        else
        {
            Console.WriteLine("El nombre no puede estar vacío.");
        }
    }
}

// Propiedad con validación: matrícula de al menos 3 caracteres.
public string Matricula
{
    get { return matricula; }
    set
    {
        if (value != null && value.Length >= 3)
        {
            matricula = value;
        }
        else
        {
            Console.WriteLine("La matrícula debe tener al menos 3 caracteres.");
        }
    }
}

// Auto-propiedad: la carrera no necesita validación.
public string Carrera { get; set; }

// Solo lectura: calcula el promedio al momento.
public double Promedio
{
    get
    {
        if (totalCalifs == 0) return 0;
        int suma = 0;
        for (int i = 0; i < totalCalifs; i++) suma += calificaciones[i];
        return (double)suma / totalCalifs;
    }
}

// El constructor usa las propiedades: valida al nacer.
public Estudiante(string nombre, string matricula, string carrera)
{

```

```

        Nombre = nombre;
        Matricula = matricula;
        Carrera = carrera;
    }

    public void AgregarCalificacion(int cal)
    {
        if (totalCalifs < calificaciones.Length)
        {
            calificaciones[totalCalifs] = cal;
            totalCalifs++;
        }
    }

    public void Presentarse()
    {
        Console.WriteLine("Soy " + Nombre + " (" + Matricula + "), " +
                           Carrera + ", promedio: " + Promedio);
    }
}

class Program
{
    static void Main()
    {
        Estudiante e = new Estudiante("Laura", "A123", "Ingeniería");
        e.AgregarCalificacion(95);
        e.AgregarCalificacion(88);
        e.Presentarse();

        // Intentos inválidos: el objeto se defiende.
        e.Nombre = ""; // rechazado
        e.Matricula = "X"; // rechazado: muy corta
        e.Presentarse(); // datos intactos
    }
}

```

Explicación: cada campo privado tiene su propiedad pública con reglas. `value.Trim() != ""` evita nombres que sean solo espacios en blanco: un detalle profesional. La matrícula exige mínimo 3 caracteres. La auto-propiedad Carrera muestra cuándo usar la forma corta. Promedio es de solo lectura y calcula al vuelo: siempre refleja las calificaciones actuales sin almacenar un valor que podría desactualizarse. El constructor asigna mediante propiedades (`Nombre = nombre`), así que un Estudiante nunca nace con nombre vacío ni matrícula corta. En Main, los intentos inválidos se rechazan y el objeto conserva su estado: encapsulamiento funcionando.

Solución 2: termómetro con validación. Rango físico real y propiedad derivada.

```
using System;

class Termometro
{
    double temperaturaCelsius;

    // Solo acepta valores físicamente posibles.
    public double TemperaturaCelsius
    {
        get { return temperaturaCelsius; }
        set
        {
            if (value >= -273.15 && value <= 1000)
            {
                temperaturaCelsius = value;
            }
            else
            {
                Console.WriteLine("Temperatura fuera de rango físico. No se cambió.");
            }
        }
    }

    // Propiedad derivada de solo lectura: convierte automáticamente.
    public double TemperaturaFahrenheit
    {
        get { return temperaturaCelsius * 9.0 / 5.0 + 32; }
    }

    public Termometro(double inicial)
    {
        TemperaturaCelsius = inicial; // valida al nacer
    }
}

class Program
{
    static void Main()
    {
        Termometro t = new Termometro(25);
        Console.WriteLine(t.TemperaturaCelsius + " C = " +
            t.TemperaturaFahrenheit + " F"); // 25 C = 77 F

        t.TemperaturaCelsius = -300; // rechazado: bajo el cero absoluto
    }
}
```



```

        Console.WriteLine("Sigue en: " + t.TemperaturaCelsius + " C");

        t.TemperaturaCelsius = 100; // aceptado
        Console.WriteLine(t.TemperaturaCelsius + " C = " +
                          t.TemperaturaFahrenheit + " F"); // 100 C = 212 F
    }
}

```

Explicación: el rango -273.15 a 1000 viene de la física real: nada puede estar más frío que el cero absoluto. La propiedad TemperaturaFahrenheit no almacena nada: calcula la conversión cada vez que se lee, así que nunca se desincroniza de los Celsius. Fíjate en 9.0 / 5.0 con decimales: si escribiras 9 / 5 con enteros, C# daría 1 (división entera truncada) y la conversión saldría mal. Este es un bug clásico que vale oro conocer. El constructor valida al nacer usando la propiedad, y en Main vemos el rechazo del valor imposible (-300) sin que el objeto se corrompa.

Solución 3: jerarquía de animales. Herencia y polimorfismo básicos con la prueba de sustitución.

```

using System;

class Animal
{
    public string Nombre { get; set; }

    public Animal(string nombre)
    {
        Nombre = nombre;
    }

    // virtual: cada animal hará su propio sonido.
    public virtual void HacerSonido()
    {
        Console.WriteLine(Nombre + " hace un sonido genérico.");
    }
}

class Perro : Animal
{
    public Perro(string nombre) : base(nombre) { }

    public override void HacerSonido()
    {
        Console.WriteLine(Nombre + " dice: ¡Guau!");
    }
}

```

```

        // Método propio: solo los perros mueven la cola.
        public void MoverCola()
        {
            Console.WriteLine(Nombre + " mueve la cola feliz.");
        }
    }

class Gato : Animal
{
    public Gato(string nombre) : base(nombre) { }

    public override void HacerSonido()
    {
        Console.WriteLine(Nombre + " dice: ¡Miau!");
    }

    public void Ronronear()
    {
        Console.WriteLine(Nombre + " ronronea...");
    }
}

class Program
{
    static void Main()
    {
        Animal[] animales = new Animal[2];
        animales[0] = new Perro("Firulais");
        animales[1] = new Gato("Michi");

        // Polimorfismo: el mismo mensaje, cada animal responde a su manera.
        foreach (Animal a in animales)
        {
            a.HacerSonido();
        }

        // Sustitución: el método pide Animal, recibe Perro y Gato.
        Alimentar(new Perro("Rex"));
        Alimentar(new Gato("Luna"));

        // Los métodos propios se usan con referencias del tipo concreto.
        Perro p = new Perro("Solovino");
        p.MoverCola(); // MoverCola no existe en Animal: necesitas un Perro.
    }
}

```

```

static void Alimentar(Animal a)
{
    Console.WriteLine("Alimentando a " + a.Nombre);
    a.HacerSonido(); // ¡también es polimórfico aquí!
}
}

```

Explicación: Animal define lo común (Nombre) y el método virtual HacerSonido con una versión genérica por defecto. Cada hija llama a : base(nombre) en su constructor para inicializar lo heredado, hace override del sonido y agrega su método propio. En Main, el arreglo Animal[] demuestra la sustitución: guarda un Perro y un Gato porque ambos SON animales. El foreach polimórfico imprime "¡Guau!" y "¡Miau!" con el mismo código. El método Alimentar recibe Animal y acepta cualquier hija, y dentro de él la llamada a.HacerSonido() también es polimórfica: el despacho dinámico funciona en cualquier parte, no solo en ciclos. La última parte muestra el límite: para usar MoverCola() necesitas una referencia de tipo Perro, porque la base Animal no conoce ese método. Esto es importante: el polimorfismo te da lo común a través de la base; lo específico requiere el tipo concreto.

Solución 4: figuras con perímetro. Extendemos el contrato abstracto con un segundo método.

```

using System;

abstract class Figura
{
    public string Color { get; set; }

    public abstract double CalcularArea();
    // Nuevo contrato: toda figura también sabe su perímetro.
    public abstract double CalcularPerimetro();
}

class Circulo : Figura
{
    double radio;
    public Circulo(double radio, string color)
    {
        this.radio = radio;
        this.Color = color;
    }
    public override double CalcularArea()
    {
        return Math.PI * radio * radio;
    }
}

```

```

    public override double CalcularPerimetro()
    {
        return 2 * Math.PI * radio;
    }
}

class Cuadrado : Figura
{
    double lado;
    public Cuadrado(double lado, string color)
    {
        this.lado = lado;
        this.Color = color;
    }
    public override double CalcularArea()
    {
        return lado * lado;
    }
    public override double CalcularPerimetro()
    {
        return 4 * lado;
    }
}

class Triangulo : Figura
{
    double lado; // equilátero: los tres lados iguales
    double altura;
    public Triangulo(double lado, double altura, string color)
    {
        this.lado = lado;
        this.altura = altura;
        this.Color = color;
    }
    public override double CalcularArea()
    {
        return (lado * altura) / 2;
    }
    public override double CalcularPerimetro()
    {
        return 3 * lado;
    }
}

class Program

```

```

{
    static void Main()
    {
        Figura[] figuras = new Figura[3];
        figuras[0] = new Circulo(5, "rojo");
        figuras[1] = new Cuadrado(4, "azul");
        figuras[2] = new Triangulo(6, 5.2, "verde");

        // Un solo ciclo imprime área Y perímetro de cada figura.
        // Ambos métodos son polimórficos.
        foreach (Figura f in figuras)
        {
            Console.WriteLine("Figura " + f.Color +
                               ": área = " + f.CalcularArea() +
                               ", perímetro = " + f.CalcularPerimetro());
        }
    }
}

```

Explicación: al agregar `CalcularPerimetro()` como abstracto, el compilador obliga a TODAS las hijas a implementarlo: si olvidas el override en alguna, el código no compila. Esa es la fuerza del contrato abstracto: es imposible que una figura "se le olvide" saber su perímetro. Para el triángulo asumimos equilátero para simplificar (en un sistema real pedirías los tres lados). En `Main`, un solo `foreach` obtiene área y perímetro de cada figura sin saber de qué tipo es cada una. Observa cómo agregar un método abstracto a la base mejora todo el sistema de golpe: cada figura gana la capacidad automáticamente al implementar su versión. Este es el poder de diseñar con abstracciones.

Solución 5: empleados con bono polimórfico. Nómina que funciona con cualquier tipo de empleado.

```

using System;

class Empleado
{
    public string Nombre { get; set; }
    public double SalarioBase { get; set; }

    public Empleado(string nombre, double salarioBase)
    {
        Nombre = nombre;
        SalarioBase = salarioBase;
    }

    // Versión por defecto: el pago es el salario base entre 12.

```

```

    public virtual double CalcularPagoMensual()
    {
        return SalarioBase / 12;
    }
}

class Vendedor : Empleado
{
    public double ComisionMensual { get; set; }

    public Vendedor(string nombre, double salarioBase, double comision)
        : base(nombre, salarioBase)
    {
        ComisionMensual = comision;
    }

    public override double CalcularPagoMensual()
    {
        // Reutiliza la base y suma lo propio: no duplica la división.
        return base.CalcularPagoMensual() + ComisionMensual;
    }
}

class Gerente : Empleado
{
    public double BonoMensual { get; set; }

    public Gerente(string nombre, double salarioBase, double bono)
        : base(nombre, salarioBase)
    {
        BonoMensual = bono;
    }

    public override double CalcularPagoMensual()
    {
        return base.CalcularPagoMensual() + BonoMensual;
    }
}

class Program
{
    static void Main()
    {
        Empleado[] personal = new Empleado[3];
        personal[0] = new Vendedor("Ana", 240000, 15000);
    }
}

```

```

        personal[1] = new Vendedor("Beto", 240000, 8000);
        personal[2] = new Gerente("Carmen", 600000, 20000);

        double nominaTotal = 0;
        foreach (Empleado e in personal)
        {
            // Polimorfismo: cada empleado calcula SU pago mensual.
            double pago = e.CalcularPagoMensual();
            Console.WriteLine(e.Nombre + ": $" + pago);
            nominaTotal += pago;
        }
        Console.WriteLine("Nómina total del mes: $" + nominaTotal);
    }
}

```

Explicación: la base Empleado define el cálculo por defecto y lo marca virtual. Cada hija hace override reutilizando base.CalcularPagoMensual() y sumando su extra: así la lógica de "salario entre 12" vive en un solo lugar y no se duplica. Si mañana cambia la forma de calcular el salario base mensual, se cambia una vez en la base y todas las hijas heredan el cambio. En Main, la nómina se calcula con un foreach polimórfico: Ana cobra 20000 + 15000 = 35000, Beto 20000 + 8000 = 28000, Carmen 50000 + 20000 = 70000; total 133000. El código de nómina no conoce a Vendedor ni a Gerente: trabaja con Empleado y el polimorfismo resuelve el resto. Si el próximo mes contratas un Practicante con su propia regla de pago, la nómina lo acepta sin cambios.

Solución 6: clase sellada de utilidades. sealed para un diseño terminado y excepciones en validaciones críticas.

```

using System;

// sealed: esta clase es una herramienta terminada.
// Nadie debe heredarla para "extenderla": si necesitas más
// operaciones, crea otra clase. Heredar una calculadora para
// agregarle métodos es el mal uso de herencia del que hablamos.
sealed class CalculadoraSegura
{
    public double Sumar(double a, double b)
    {
        return a + b;
    }

    public double Restar(double a, double b)
    {
        return a - b;
    }
}

```

```

    }

    public double Multiplicar(double a, double b)
    {
        return a * b;
    }

    public double Dividir(double a, double b)
    {
        // Validación crítica: dividir entre cero es un error
        // matemático, así que lanzamos una excepción con mensaje claro
        // en lugar de devolver un valor silenciosamente incorrecto.
        if (b == 0)
        {
            throw new ArgumentException("No se puede dividir entre cero.");
        }
        return a / b;
    }
}

class Program
{
    static void Main()
    {
        CalculadoraSegura calc = new CalculadoraSegura();

        Console.WriteLine("10 + 5 = " + calc.Sumar(10, 5));
        Console.WriteLine("10 - 5 = " + calc.Restar(10, 5));
        Console.WriteLine("10 * 5 = " + calc.Multiplicar(10, 5));
        Console.WriteLine("10 / 5 = " + calc.Dividir(10, 5));

        // La división entre cero se captura con try/catch:
        // el programa no se rompe, muestra el mensaje y sigue.
        try
        {
            Console.WriteLine("10 / 0 = " + calc.Dividir(10, 0));
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error capturado: " + ex.Message);
        }

        Console.WriteLine("El programa sigue funcionando.");
    }
}

```


Explicación: la clase es sealed porque representa una herramienta completa y cerrada: no hay una relación "es un" natural que justifique heredarla (¿qué sería una "calculadora segura especializada"?). Si alguien quiere más operaciones, compone o crea otra clase; heredar para agregar métodos sueltos es el abuso de herencia que queremos evitar. El método Dividir valida el divisor y lanza ArgumentException con un mensaje claro cuando es cero: esto es "fallar rápido", mucho mejor que devolver infinito o cero silenciosamente y que el error aparezca diez pasos después. En Main, el try/catch demuestra cómo el código cliente maneja la excepción con elegancia: la captura, muestra el mensaje y el programa continúa. Nota que instanciamos la clase con new aunque sea sealed: sealed impide heredar, no impide crear objetos.

Solución 7: composición vs herencia. El Coche TIENE UN Motor, no ES UN motor.

```
using System;

// El Motor es una pieza independiente con su propia responsabilidad.
class Motor
{
    bool encendido = false;

    public void Encender()
    {
        encendido = true;
        Console.WriteLine("Motor encendido.");
    }

    public void Apagar()
    {
        encendido = false;
        Console.WriteLine("Motor apagado.");
    }

    public bool EstaEncendido()
    {
        return encendido;
    }
}

// El Coche TIENE UN Motor (composición): lo guarda como campo privado
// y lo usa en sus propios métodos. No hereda de Motor.
class Coche
{
    string modelo;
    // Composición: el coche contiene un motor, no ES un motor.
```

```

    Motor motor = new Motor();

    public Coche(string modelo)
    {
        this.modelo = modelo;
    }

    public void Arrancar()
    {
        Console.WriteLine("Arrancando el " + modelo + "...");
        motor.Encender(); // delega en su pieza interna
    }

    public void Detener()
    {
        Console.WriteLine("Deteniendo el " + modelo + "...");
        motor.Apagar();
    }
}

class Program
{
    static void Main()
    {
        Coche miCoche = new Coche("Tsuru 2005");
        miCoche.Arrancar();
        miCoche.Detener();

        // ¿Por qué NO herencia aquí? La prueba del "es un":
        // "Un Coche ES UN Motor" suena absurdo. Un coche TIENE un motor,
        // igual que TIENE ruedas y TIENE puertas. La relación real es
        // de CONTENIDO (composición), no de ESPECIALIZACIÓN (herencia).
        //
        // Si hiciéramos class Coche : Motor, el coche heredaría Encender()
        // y Apagar() como si ÉL fuera el motor, y cualquier cambio en Motor
        // afectaría a Coche de formas impredecibles. Además, ¿qué pasa
        // cuando el coche también "necesita" ser un Chasis o unas Ruedas?
        // C# no permite herencia múltiple, pero la composición sí permite
        // tener todas las piezas: Motor, 4 Ruedas, Chasis, etc.
    }
}

```

Explicación: la clase Coche declara un campo privado `motor = new Motor()`: el coche nace con su motor incluido, creado automáticamente. Los métodos `Arrancar` y `Detener` delegan en el motor interno: el mundo exterior le habla al coche, y el coche le habla a sus piezas. Esta es la

esencia de la composición: objetos que contienen otros objetos y coordinan su trabajo. El comentario en Main explica por qué la herencia sería incorrecta: no pasa la prueba del "es un", acoplaría rígidamente ambas clases e impediría tener más piezas (C# solo permite un padre). Regla para recordar: "es un" = herencia, "tiene un" = composición. La mayoría de las relaciones del mundo real son "tiene un", por eso los profesionales favorecen la composición.

Solución 8: jerarquía de cuentas bancarias. Clase abstracta con protected y dos hijas con intereses distintos.

```
using System;

// Clase abstracta: "cuenta bancaria en general" es un concepto,
// las cuentas reales son las hijas concretas.
abstract class CuentaBancaria
{
    public string NumeroCuenta { get; set; }

    // protected: las hijas lo usan directamente, el mundo exterior no.
    protected double saldo;

    // Propiedad pública de solo lectura para consultar el saldo.
    public double Saldo
    {
        get { return saldo; }
    }

    public CuentaBancaria(string numeroCuenta, double saldoInicial)
    {
        NumeroCuenta = numeroCuenta;
        saldo = saldoInicial;
    }

    // Método concreto con validación: todas las cuentas depositan igual.
    public void Depositar(double cantidad)
    {
        if (cantidad > 0)
        {
            saldo += cantidad;
            Console.WriteLine("Depósito de $" + cantidad + " en " + NumeroCuenta);
        }
        else
        {
            Console.WriteLine("Cantidad inválida.");
        }
    }
}
```

```

// Contrato: cada tipo de cuenta calcula su interés a su manera.
public abstract double CalcularInteresMensual();

public void Mostrar()
{
    Console.WriteLine("Cuenta " + NumeroCuenta + ", saldo: $" + saldo);
}
}

class CuentaAhorros : CuentaBancaria
{
    public CuentaAhorros(string numeroCuenta, double saldoInicial)
        : base(numeroCuenta, saldoInicial) { }

    // Ahorros: 0.5% mensual sobre el saldo.
    public override double CalcularInteresMensual()
    {
        return saldo * 0.005;
    }
}

class CuentaCredito : CuentaBancaria
{
    // Crédito: el saldo puede ser negativo hasta -10000.
    public CuentaCredito(string numeroCuenta, double saldoInicial)
        : base(numeroCuenta, saldoInicial) { }

    // En crédito, el interés del 2% se cobra sobre la deuda.
    // Usamos el valor absoluto porque el saldo deudor es negativo.
    public override double CalcularInteresMensual()
    {
        if (saldo < 0)
        {
            return -saldo * 0.02; // interés a pagar (positivo)
        }
        return 0; // sin deuda, sin interés
    }

    // Método propio: usar la tarjeta aumenta la deuda, con límite.
    public void UsarTarjeta(double cantidad)
    {
        if (saldo - cantidad >= -10000)
        {
            saldo -= cantidad;
        }
    }
}

```

```

        Console.WriteLine("Cargo de $" + cantidad + ". Deuda: $" + (-saldo));
    }
    else
    {
        Console.WriteLine("Límite de crédito excedido.");
    }
}
}

class Program
{
    static void Main()
    {
        CuentaBancaria[] cuentas = new CuentaBancaria[2];
        cuentas[0] = new CuentaAhorros("AH-001", 50000);
        cuentas[1] = new CuentaCredito("TC-001", 0);

        // La de crédito usa su método propio (referencia concreta).
        CuentaCredito credito = (CuentaCredito)cuentas[1];
        credito.UsarTarjeta(8000); // deuda de 8000

        // Polimorfismo: el mismo mensaje, cada cuenta calcula su interés.
        foreach (CuentaBancaria c in cuentas)
        {
            c.Mostrar();
            Console.WriteLine("Interés mensual: $" + c.CalcularInteresMensual());
            Console.WriteLine();
        }
        // Ahorros: 50000 * 0.005 = 250. Crédito: 8000 * 0.02 = 160.
    }
}

```

Explicación: la clase abstracta CuentaBancaria combina todo lo aprendido: el campo saldo es protected (las hijas lo manipulan, el mundo solo lo lee vía la propiedad Saldo), Depositar es un método concreto con validación que todas heredan igual, y CalcularInteresMensual es abstracto porque cada tipo de cuenta tiene su propia regla. CuentaAhorros aplica 0.5% sobre el saldo; CuentaCredito aplica 2% sobre la deuda (el valor absoluto del saldo negativo) y agrega su método propio UsarTarjeta con validación del límite de -10000. En Main, el arreglo de la base guarda ambas cuentas y el foreach polimórfico calcula cada interés correctamente: 250 para ahorros, 160 para crédito. Para usar el método propio UsarTarjeta necesitamos una referencia de tipo CuentaCredito (conversión explícita), lo que refuerza la lección: lo común va por la base, lo específico por el tipo concreto. Este diseño es exactamente como los bancos reales modelan sus productos: una abstracción común, reglas específicas por producto.

Resumen de la unidad

- El encapsulamiento esconde los detalles internos del objeto y expone solo una interfaz pública controlada: es el primer pilar de la POO.
- `public` es visible para todos, `private` solo dentro de la propia clase, `protected` dentro de la clase y sus hijas; por defecto los miembros son `private`.
- Regla de oro: campos siempre privados; el acceso exterior se da con propiedades y métodos públicos.
- Una propiedad se usa como un campo desde afuera pero ejecuta código por dentro: `get` para leer, `set` para escribir, con el valor entrante en la variable `value`.
- La validación en los `setters` hace que las reglas se cumplan automáticamente en cada asignación, sin depender de que el programador se acuerde de validar.
- Las auto-propiedades (`get`; `set`;) son la forma corta cuando no se necesita validación; el compilador crea el campo oculto.
- Las propiedades pueden ser de solo lectura (solo `get`), y el `set` puede ser privado para permitir lectura pública con escritura interna.
- Cuando la validación falla puedes ignorar el valor o lanzar una excepción con `throw new ArgumentException`; fallar rápido con mensaje claro es lo profesional.
- Haz que los constructores asignen a través de las propiedades para que la validación también proteja el nacimiento del objeto.
- La herencia permite que una clase hija reutilice los miembros de una clase base y agregue los suyos: es el segundo pilar de la POO.
- Se declara con dos puntos (`class Gerente : Empleado`) y la prueba de validez es la frase "la hija ES UNA base": si no suena natural, no uses herencia.
- La palabra `base` permite llamar al constructor o a miembros de la clase padre; el constructor de la hija debe inicializar la parte heredada con `: base(...)`.
- `protected` comparte miembros con las hijas sin exponerlos al mundo exterior: es el nivel de acceso diseñado para la herencia.
- C# solo permite herencia simple (un solo padre por clase); las jerarquías deben mantenerse poco profundas, dos o tres niveles como máximo.
- Favorece la composición ("tiene un") sobre la herencia ("es un"): la composición es más flexible y evita jerarquías rígidas y artificiales.

- El polimorfismo permite tratar objetos de distintas hijas de forma uniforme a través de la base, y cada objeto responde con su propio comportamiento: es el tercer pilar.
- virtual en la base y override en la hija trabajan en pareja: sin ambas palabras no hay polimorfismo real y se ejecuta la versión de la base.
- El despacho dinámico elige automáticamente en tiempo de ejecución la versión override del objeto real, aunque la referencia sea de la clase base.
- Una clase abstracta no permite crear objetos directos y puede tener métodos abstractos (sin cuerpo) que obligan a cada hija concreta a implementarlos con override.
- Usa clases abstractas cuando la base sea un concepto general sin implementación por defecto sensata; usa virtual cuando sí haya un comportamiento base razonable.
- sealed impide que una clase sea heredada (o que un método override se redefina más abajo): sirve para cerrar diseños terminados.
- Errores típicos: campos públicos sin validación, olvidar virtual/override, usar herencia solo para reutilizar código, no llamar a base en constructores, validar solo en el constructor y no en el setter.
- Buenas prácticas: programa contra la clase base (recibe parámetros del tipo base), mantén jerarquías con sentido real, valida en los setters, sella lo que no deba extenderse y prefiere composición cuando la relación no sea un "es un" claro.

UNIDAD 7: Interfaces, colecciones y LINQ

Si llegaste hasta aquí, ya sabes crear clases, heredar, usar polimorfismo y organizar tu código con encapsulamiento. Eso es muchísimo: la mayoría de la gente que dice "estoy aprendiendo a programar" se queda atorada mucho antes. Pero te voy a decir algo de colega a colega: hasta ahora has estado construyendo piezas sueltas. Esta unidad es donde esas piezas empiezan a trabajar juntas como una máquina de verdad.

En esta unidad vas a ver tres temas que, combinados, son el corazón del C# profesional: las interfaces, las colecciones genéricas y LINQ. Suena a mucho, pero los tres están conectados por una misma idea: manejar datos y comportamientos de forma flexible, sin amarrarte a detalles que no te importan.

Primero, las interfaces. Piensa en esto: cuando compras un cargador para tu celular, no te importa qué marca de contactos eléctricos tiene la pared de tu casa. Te importa una sola cosa: que el enchufe entre. El enchufe es un contrato. La pared promete "te doy corriente si tu clavija tiene esta forma" y el cargador promete "mi clavija tiene esa forma". Ninguno de los dos necesita saber cómo está construido el otro por dentro. Las interfaces en C# son exactamente eso: contratos que dicen "si quieres trabajar conmigo, tienes que poder hacer estas cosas", sin decirte cómo las vas a hacer por dentro. Cuando termines la clase 19 vas a entender por qué los programadores profesionales diseñan sus programas alrededor de interfaces y no de clases concretas, y vas a poder escribir código que acepta "cualquier cosa que sepa guardarse", "cualquier cosa que se pueda comparar" o "cualquier cosa que se pueda dibujar", sin importar qué clase sea en realidad.

Segundo, las colecciones genéricas. Hasta ahora, cuando querías guardar varios datos usabas arreglos, y los arreglos tienen un problema serio: su tamaño es fijo. Es como comprar una caja de zapatos de tamaño exacto para guardar tus tenis: si compras otro par, ya no caben. En la vida real los datos crecen y se encogen: una tienda agrega productos, una agenda agrega contactos, un carrito de compras agrega y quita artículos. Las colecciones genéricas como `List<T>` y `Dictionary<TKey, TValue>` son cajas que se estiran solas. Cuando termines la clase 20 vas a saber guardar, buscar, quitar y recorrer miles de datos sin preocuparte por el tamaño, vas a elegir la colección correcta para cada situación (lista para secuencias, diccionario para búsquedas por clave), y vas a evitar los errores clásicos que truenan programas enteros, como modificar una lista mientras la recorres.

Tercero, LINQ. Este es el tema que más cara de "magia" tiene y el que más te va a servir en entrevistas de trabajo. LINQ (Language Integrated Query) te deja hacerle preguntas a tus datos con una sintaxis elegante, casi como si estuvieras escribiendo en español: "dame los

productos que cuesten más de 100, ordenados por nombre". Sin LINQ tendrías que escribir un foreach, un if adentro, otra lista para guardar resultados, ordenarlos a mano... Con LINQ es una línea. Cuando termines la clase 21 vas a filtrar, ordenar, transformar, contar, sumar y promediar colecciones con unas cuantas líneas legibles, y vas a entender las dos sintaxis de LINQ (la de métodos y la de consulta) para leer cualquier código profesional que te pongan enfrente.

¿Por qué importa todo esto en la vida real? Porque ningún programa real trabaja con tres variables sueltas. Trabaja con listas de usuarios, catálogos de productos, registros de ventas, agendas de contactos. Y ningún programa real puede darse el lujo de estar amarrado a una sola implementación: el día que tu jefe te diga "cambiamos la base de datos" o "ahora también aceptamos pagos con tarjeta además de efectivo", si tu código depende de interfaces y colecciones bien elegidas, el cambio es de horas; si no, es de semanas. Las empresas pagan bien a quien domina estos tres temas porque son la diferencia entre un programa de juguete y un sistema que aguanta el uso real.

Al terminar esta unidad podrás: definir e implementar interfaces (incluso varias en una misma clase), decidir cuándo usar una interfaz y cuándo una clase abstracta, manejar List<T> y Dictionary<TKey, TValue> con soltura (agregar, quitar, buscar, recorrer), escribir consultas LINQ con Where, Select, OrderBy, First, Count, Sum y Average en sus dos sintaxis, y armar programas completos como un catálogo de productos con filtros y reportes, y una agenda de contactos con búsqueda instantánea por nombre. En corto: vas a dejar de programar "ejercisitos" y vas a empezar a programar herramientas de verdad.

Una advertencia honesta antes de empezar: esta unidad tiene más conceptos nuevos por página que las anteriores. No la leas corriendo. Cada vez que veas un ejemplo, escríbelo tú en tu editor, ejecútalo, rómpelo a propósito (cambia algo y mira qué error sale) y luego arréglalo. Esa es la forma en que estos temas se quedan grabados. ¿Listo? Vamos.

Explicación detallada

Clase 19: Interfaces, los contratos de tu programa

Qué es una interfaz: la analogía del enchufe

Una interfaz es un contrato. No contiene código que haga cosas (bueno, casi: en versiones modernas de C# puede traer implementaciones por defecto, pero ignora eso por ahora); contiene una lista de promesas: "todo el que firme este contrato se compromete a tener estos métodos, estas propiedades, estos eventos". Fíjate en la analogía del enchufe porque es perfecta:

- La compañía de luz no sabe ni le importa qué aparato vas a conectar: una lámpara, un refrigerador, tu laptop. Solo exige: "tu clavija debe tener esta forma y soportar este voltaje".
- El fabricante del aparato no sabe ni le importa cómo genera electricidad la compañía. Solo cumple: "mi clavija tiene esa forma".

Gracias a ese contrato, millones de aparatos distintos funcionan con millones de enchufes distintos sin que nadie coordine nada. En programación pasa igual: la interfaz dice QUÉ se debe poder hacer, y cada clase decide CÓMO lo hace.

Mira la sintaxis. Las interfaces se declaran con la palabra `interface`` y por convención su nombre empieza con I mayúscula:

```
interface IVehiculo
{
    void Encender();
    void Apagar();
    int VelocidadMaxima { get; }
}
```

Eso es todo el contrato: "si eres un IVehiculo, tienes que poder encenderte, apagarte y decirme tu velocidad máxima". Nota que no hay llaves con código adentro de los métodos, solo la firma terminada en punto y coma. Es como el formato de un contrato legal: lista de cláusulas, sin explicar cómo las vas a cumplir.

Ahora, una clase "firma" el contrato con dos puntos, igual que la herencia:

```

class Automovil : IVehiculo
{
    public int VelocidadMaxima { get { return 180; } }

    public void Encender()
    {
        Console.WriteLine("El automóvil ruge al encender.");
    }

    public void Apagar()
    {
        Console.WriteLine("El automóvil se apaga.");
    }
}

class BicicletaElectrica : IVehiculo
{
    public int VelocidadMaxima { get { return 45; } }

    public void Encender()
    {
        Console.WriteLine("La bicicleta emite un bip suave.");
    }

    public void Apagar()
    {
        Console.WriteLine("La bicicleta corta la asistencia eléctrica.");
    }
}

```

Un automóvil y una bicicleta eléctrica no tienen NADA que ver entre sí: no comparten motor, ni ruedas, ni nada. No tendría sentido que heredaran de una misma clase base. Pero ambos pueden firmar el contrato `IVehiculo`. Y aquí está la magia: ahora puedes escribir métodos que reciben "cualquier `IVehiculo`" sin importar qué sea:

```

static void ProbarVehiculo(IVehiculo v)
{
    Console.WriteLine("Velocidad máxima: " + v.VelocidadMaxima);
    v.Encender();
    v.Apagar();
}

```

Ese método funciona con el automóvil, con la bicicleta eléctrica y con cualquier vehículo que alguien invente dentro de diez años, siempre que firme el contrato. Eso se llama programar contra abstracciones y es una de las habilidades que más buscan los empleadores: tu código deja de depender de cosas concretas y pasa a depender de contratos estables.

Implementar múltiples interfaces: el superpoder

Aquí viene algo que la herencia de clases NO puede hacer: una clase solo puede heredar de UNA clase base, pero puede implementar TANTAS interfaces como quiera. Es como una persona: solo tiene un padre biológico, pero puede tener muchas profesiones, muchos roles, muchas credenciales.

```
interface IVolador
{
    void Despegar();
    void Aterrizar();
}

interface INadador
{
    void Nadar();
}

class Pato : IVolador, INadador
{
    public void Despegar()
    {
        Console.WriteLine("El pato bate las alas y despega.");
    }

    public void Aterrizar()
    {
        Console.WriteLine("El pato aterriza en el lago.");
    }

    public void Nadar()
    {
        Console.WriteLine("El pato nada tranquilamente.");
    }
}
```

El pato firma dos contratos y cumple los dos. Ahora puedes pasarlo a un método que pide un IVolador ("Ilévalo al aeropuerto") o a uno que pide un INadador ("Ilévalo a la alberca"), y en

cada contexto se usa solo la parte del contrato que importa. Esto resuelve el problema clásico del "¿de quién heredo?": no heredas, firmas contratos.

En la vida real esto aparece por todos lados. Piensa en un sistema de pagos: tienes `IPagable` con el método `ProcesarPago(decimal monto)`. La tarjeta de crédito lo implementa de una forma (conectándose al banco), el efectivo de otra (abriendo la caja), los puntos de lealtad de otra (descontando puntos). Tu caja registradora solo conoce `IPagable`. El día que agreguen "pago con código QR", creas una clase nueva que firme el contrato y la caja ni se entera del cambio. Cero modificaciones al código existente. Eso, en la industria, se llama principio abierto/cerrado y vale oro.

Interfaces vs clases abstractas: cuándo usar cada una

Esta es LA pregunta de entrevista. "¿Cuál es la diferencia entre una interfaz y una clase abstracta?" Si respondes bien esto, ya vas adelante del 70 por ciento de los candidatos. Vamos por partes.

Una clase abstracta es una clase incompleta: puede tener métodos con código, campos, constructores, y también métodos abstractos (sin código) que las hijas están obligadas a implementar. Una interfaz (en su uso clásico) solo tiene firmas, nada de código.

La regla práctica para decidir:

- Usa una CLASE ABSTRACTA cuando las clases hijas comparten una identidad y comportamiento base real. Ejemplo: `Animal` como clase abstracta con el campo `nombre`, el método concreto `Dormir()` (todos los animales duermen igual en tu programa) y el método abstracto `HacerRuido()` (cada animal lo hace distinto). Hay un "es un": un Perro ES UN Animal.
- Usa una INTERFAZ cuando quieres describir una CAPACIDAD que pueden tener cosas no relacionadas. Ejemplo: `IVolador`. Un avión, un pájaro y un dron vuelan, pero no tiene sentido que hereden de una misma clase. Hay un "puede hacer": el dron PUEDE VOLAR.

Otra diferencia clave: una clase hereda de una sola base, pero implementa muchas interfaces. Si necesitas que algo tenga dos "identidades", las interfaces son tu única opción.

Míralo en una tabla mental:

- Interfaz: solo firmas (contrato puro); una clase implementa muchas; no lleva estado (campos); perfecta para capacidades entre clases no relacionadas.

- Clase abstracta: mezcla de código real y métodos abstractos; una clase hereda de una sola; sí lleva estado y constructores; perfecta para jerarquías de "es un" con comportamiento compartido.

En C# moderno (8 en adelante) las interfaces pueden traer métodos con implementación por defecto, lo que borra un poco la línea. Pero como estudiante, quédate con la regla clásica: ¿capacidad compartida entre cosas distintas? Interfaz. ¿Familia con código en común? Clase abstracta.

Interfaces que ya existen en .NET: no reinventes la rueda

.NET trae interfaces listas que las colecciones y muchos métodos esperan. Conocerlas te ahorra horas:

- `IComparable<T>`: promete el método `CompareTo`. Si tu clase lo implementa, las listas pueden ordenarse solas con `Sort()`. Ejemplo: tus productos se ordenan por precio sin que escribas el algoritmo de ordenamiento.
- `IComparable<T>`: promete `Equals` tipado. Sirve para que `Contains` y `Dictionary` comparen tus objetos correctamente.
- `IEnumerable<T>`: promete "me puedes recorrer con foreach". Todas las colecciones la implementan; los métodos LINQ trabajan sobre ella.
- `IDisposable`: promete el método `Dispose()`. Es el contrato de "tengo recursos que hay que liberar" (archivos, conexiones). Se usa con `using`, que verás a fondo en la unidad 8.

Ejemplo rápido de `IComparable` para que veas el poder:

```
class Producto : IComparable<Producto>
{
    public string Nombre { get; set; }
    public decimal Precio { get; set; }

    // CompareTo devuelve negativo si este va antes, cero si empatan, positivo si
    va después
    public int CompareTo(Producto otro)
    {
        return this.Precio.CompareTo(otro.Precio);
    }
}
```

Con eso, `listaDeProductos.Sort()` ordena por precio automáticamente. Tú solo definiste la regla de comparación; .NET puso el algoritmo. Esa es la filosofía de las interfaces: tú pones las reglas de tu negocio, la plataforma pone la maquinaria.

Errores comunes con interfaces (y cómo evitarlos)

1. Intentar crear un objeto de una interfaz: `IVehiculo v = new IVehiculo();` NO compila. Una interfaz es un contrato, no una cosa. Lo correcto es `IVehiculo v = new Automovil();`: la variable es del tipo contrato, el objeto es de la clase concreta.
2. Olvidar implementar un miembro: si tu clase dice que implementa `IVehiculo` pero no trae el método `Apagar()`, el compilador te lo reclama. Léelo como un abogado revisando el contrato: "firmaste, ahora cumple todo". El error del compilador te dice exactamente qué miembro falta.
3. Declarar los miembros de la interfaz como `public` dentro de la interfaz (en el estilo clásico): en la declaración de la interfaz no se pone modificador de acceso en cada miembro; por defecto todo es público. (En C# moderno sí se puede, pero si estás empezando, no lo pongas y evitas confusiones.)
4. Confundir "implementar" con "heredar" en el diseño: si pones TODO en una clase base enorme ("clase dios"), cada cambio la rompe toda. Divide responsabilidades en interfaces pequeñas y específicas. Una interfaz con 20 métodos es una mala interfaz; tres interfaces de 5 métodos son mejores. Esto tiene nombre: principio de segregación de interfaces.

Clase 20: Colecciones genéricas, cajas que se estiran solas

El problema de los arreglos

Los arreglos que ya conoces funcionan, pero tienen dos limitaciones que en programas reales duelen:

1. Tamaño fijo: `new string[10]` guarda 10 y ni uno más. Si llega el onceavo dato, truena o tienes que crear otro arreglo más grande y copiar todo a mano.
2. Operaciones incómodas: quitar un elemento del medio, insertar al principio o buscar algo requiere código manual con ciclos.

Las colecciones genéricas resuelven esto. La palabra "genéricas" significa que funcionan con cualquier tipo de dato: `List<string>`, `List<int>`, `List<Producto>`. La T entre < > es un marcador que dice "aquí va el tipo que tú me digas". Es como un molde: el mismo molde sirve para gelatinas de fresa, de limón o de uva; tú decides el sabor al usarlo.

List<T>: tu lista de todos los días

`List<T>` es la colección que más vas a usar en tu vida profesional. Es una secuencia ordenada que crece sola. Está en el espacio de nombres `System.Collections.Generic`.

```
using System.Collections.Generic;

List<string> invitados = new List<string>();

// Agregar al final
invitados.Add("Ana");
invitados.Add("Beto");
invitados.Add("Carla");

// Insertar en una posición
invitados.Insert(1, "Diana"); // Ana, Diana, Beto, Carla

// Acceder por índice, igual que un arreglo
Console.WriteLine(invitados[0]); // Ana

// Saber cuántos hay
Console.WriteLine(invitados.Count); // 4

// Preguntar si existe
bool vieneBeto = invitados.Contains("Beto"); // true

// Quitar por valor o por posición
invitados.Remove("Diana"); // quita la primera aparición
invitados.RemoveAt(0);      // quita el de la posición 0

// Recorrer
foreach (string nombre in invitados)
{
    Console.WriteLine("- " + nombre);
}
```

Métodos que debes memorizar porque los usarás a diario: `Add`, `Insert`, `Remove`, `RemoveAt`, `Clear` (vacía la lista), `Contains`, `IndexOf` (posición de un valor, o -1 si no está), `Count` (propiedad, no método), `Sort()` (ordena si el tipo sabe compararse), `Reverse()`, `ToArray()` (convierte a arreglo cuando lo necesites).

`List<T>` guarda el orden en que agregaste las cosas y permite duplicados. Es tu opción por defecto cuando piensas "necesito guardar varios X en orden". Carrito de compras, lista de alumnos, historial de movimientos: todo eso es `List<T>`.

Dictionary<TKey, TValue>: búsqueda instantánea por clave

Ahora imagina otro problema: tienes 10,000 contactos y quieres encontrar el teléfono de "María González". Con una lista tendrías que recorrer uno por uno hasta encontrarla. Con un diccionario, la encuentras al instante. ¿Por qué? Porque un diccionario no guarda "una secuencia", guarda pares CLAVE -> VALOR, como un diccionario de verdad (palabra -> definición) o como una agenda telefónica (nombre -> teléfono).

```
Dictionary<string, string> telefonos = new Dictionary<string, string>();

// Agregar pares clave-valor
telefonos.Add("Ana", "555-0101");
telefonos.Add("Beto", "555-0202");

// Otra forma de agregar (o actualizar si ya existe)
telefonos["Carla"] = "555-0303";

// Leer por clave: rapidísimo, no importa si hay un millón
Console.WriteLine(telefonos["Ana"]); // 555-0101

// Preguntar si existe la clave ANTES de leer (importantísimo)
if (telefonos.ContainsKey("Diana"))
{
    Console.WriteLine(telefonos["Diana"]);
}
else
{
    Console.WriteLine("Diana no está en la agenda.");
}

// Quitar
telefonos.Remove("Beto");

// Recorrer todos los pares
foreach (var par in telefonos)
{
    Console.WriteLine(par.Key + " -> " + par.Value);
}

// Solo claves o solo valores
foreach (string nombre in telefonos.Keys) { Console.WriteLine(nombre); }
foreach (string numero in telefonos.Values) { Console.WriteLine(numero); }
```

Dos reglas de oro del diccionario:

1. Las CLAVES no se repiten. Si intentas `Add` con una clave que ya existe, truena con una excepción. Por eso existe `ContainsKey`: pregunta antes. O usa el indexador `telefonos["Carla"] = ...`, que agrega si no existe y actualiza si existe.
2. Nunca leas `diccionario[clave]` sin estar seguro de que la clave existe. Si no existe, lanza `KeyNotFoundException` y tu programa se cae. La forma profesional es `TryGetValue`, que intenta y te dice si pudo:

```
string numero;
if (telefonos.TryGetValue("Elena", out numero))
{
    Console.WriteLine("El número de Elena es " + numero);
}
else
{
    Console.WriteLine("No tenemos el número de Elena.");
}
```

`TryGetValue` es el patrón que verás en código profesional una y otra vez: intenta obtener el valor; si la clave existe devuelve true y pone el valor en la variable `out`; si no, devuelve false y no truena nada. Apréndetelo de memoria.

¿Cuándo usar cada una? Regla simple:

- `List<T>`: "tengo varios elementos en orden y los recorro o los proceso en secuencia". Permite duplicados. Ejemplos: líneas de un ticket, pasos de una receta, historial.
- `Dictionary<TKey, TValue>`: "necesito encontrar algo RÁPIDO por un identificador único". Claves únicas. Ejemplos: agenda (nombre -> teléfono), catálogo (código -> producto), configuraciones (nombre de opción -> valor), conteos (palabra -> veces que aparece).

Y para que sepas que existen (las verás en código ajeno): `HashSet<T>` (como una lista pero sin duplicados y con búsquedas rápidas), `Queue<T>` (cola: el primero en entrar es el primero en salir, como la fila del banco), `Stack<T>` (pila: el último en entrar es el primero en salir, como una pila de platos).

El error que truena programas: modificar la lista dentro del foreach

Este error es tan común que merece su propia sección con letras grandes mentales. Mira:

```
// MAL: esto lanza InvalidOperationException
foreach (string nombre in invitados)
{
    if (nombre == "Beto")
    {
        invitados.Remove(nombre); // ¡Modificando mientras recorres!
    }
}
```

¿Por qué truena? Porque `foreach` usa un "enumerador" que lleva la cuenta de dónde va; si cambias la lista a la mitad, el enumerador se pierde y .NET prefiere tronar que darte resultados corruptos. Es como intentar contar las sillas de un salón mientras alguien las mueve: el conteo sale mal.

Formas correctas de hacerlo:

```
// Opción 1: recorre una COPIA y modifica la original
foreach (string nombre in invitados.ToList())
{
    if (nombre.StartsWith("B"))
    {
        invitados.Remove(nombre);
    }
}

// Opción 2: recorre hacia ATRÁS con for (los índices no se desfasan)
for (int i = invitados.Count - 1; i >= 0; i--)
{
    if (invitados[i].StartsWith("B"))
    {
        invitados.RemoveAt(i);
    }
}

// Opción 3: la forma elegante con LINQ (la verás en la clase 21)
invitados.RemoveAll(n => n.StartsWith("B"));
```

La opción 3 te adelanta algo: LINQ y los métodos modernos de las colecciones existen justo para que no escribas estos ciclos peligrosos a mano.

Otro error clásico: usar índices fuera de rango. `lista[lista.Count]` truena porque los índices van de 0 a Count - 1. Y con arreglos era igual. La diferencia es que con listas dinámicas el Count cambia, así que verifica antes de indexar a ciegas, sobre todo después de quitar elementos.

Más poder de List<T>: Find, Exists y ForEach

Además de lo básico, `List<T>` trae métodos de búsqueda con predicados (condiciones lambda) que te ahorran ciclos manuales:

```
List<string> nombres = new List<string> { "Ana", "Beto", "Carla", "Bruno" };

// Find: devuelve el PRIMER elemento que cumpla, o null si ninguno
string conB = nombres.Find(n => n.StartsWith("B")); // "Beto"

// FindAll: devuelve UNA LISTA con todos los que cumplan
List<string> todosConB = nombres.FindAll(n => n.StartsWith("B")); // Beto, Bruno

// Exists: pregunta si AL MENOS UNO cumple (como Any de LINQ, pero propio de la
// lista)
bool hayAnas = nombres.Exists(n => n == "Ana"); // true

// TrueForAll: ¿TODOS cumplen?
bool todosLargos = nombres.TrueForAll(n => n.Length >= 3); // true

// ForEach: ejecuta una acción por cada elemento (método, no la palabra reservada
// foreach)
nombres.ForEach(n => Console.WriteLine("Hola, " + n));

// ConvertAll: transforma cada elemento y devuelve una lista nueva del otro tipo
List<int> longitudes = nombres.ConvertAll(n => n.Length); // 3, 4, 5, 5
```

¿Cuál es la diferencia entre `Find` y `Where(...).FirstOrDefault`? Casi ninguna en el resultado; `Find` es un método propio de `List<T>` (más directo, no necesita `using System.Linq`), mientras que LINQ funciona sobre cualquier colección. En código profesional verás ambos. Mi consejo: si ya estás en "modo LINQ" en esa parte del programa, usa LINQ para mantener un estilo consistente; si solo necesitas una búsqueda rápida en una lista, `Find` es perfectamente válido.

Un detalle de rendimiento que vale oro: `List<T>` por dentro usa un arreglo que se agranda automáticamente. Cuando se llena, crea uno al doble de tamaño y copia todo. Eso significa que agregar al final es rapidísimo casi siempre, pero de vez en cuando hay un "brinco" de copia. Si sabes de antemano que guardarás 10,000 elementos, puedes decirle la capacidad inicial: `new List<int>(10000)`. Evitas los brincos y el programa corre más fluido. Es un micro-detalle, pero es el tipo de cosa que distingue a quien entiende la herramienta de quien solo la usa.

Clase 21: LINQ, pregúntale cosas a tus datos

Qué es LINQ y por qué parece magia

LINQ son las siglas de Language Integrated Query: consultas integradas al lenguaje. La idea es revolucionaria por lo simple: en vez de escribir ciclos a mano para filtrar, ordenar o transformar datos, escribes una CONSULTA que dice qué quieres, y LINQ se encarga del cómo.

Piensa en una bibliotecaria experta. Tú le dices: "quiero los libros de ciencia ficción publicados después del 2010, ordenados por autor". Tú no le dices "ve al pasillo 3, revisa cada libro, fíjate en la etiqueta del género, compara el año...". Solo describes el resultado que quieres. LINQ es esa bibliotecaria: tú describes, ella ejecuta.

Sin LINQ, filtrar productos caros se ve así:

```
List<Producto> caros = new List<Producto>();
foreach (Producto p in productos)
{
    if (p.Precio > 100)
    {
        caros.Add(p);
    }
}
```

Con LINQ:

```
using System.Linq;

var caros = productos.Where(p => p.Precio > 100).ToList();
```

Una línea contra siete. Y no es solo menos código: es código que se lee como la intención ("productos donde el precio sea mayor a 100"). Cuando un programa crece a miles de líneas, esa legibilidad es la diferencia entre un código mantenible y una pesadilla.

Para usar LINQ necesitas `using System.Linq;` arriba de tu archivo. Los métodos LINQ trabajan sobre cualquier colección que implemente `IEnumerable<T>`: listas, arreglos, diccionarios (sus pares), hasta resultados de bases de datos.

Las expresiones lambda: el ingrediente secreto

Ese `p => p.Precio > 100` se llama expresión lambda. Léelo así: "dado un p, evalúa si p.Precio > 100". La flecha `=>` se lee "tal que" o "dado que devuelve". Es una función pequeña, anónima, escrita en línea.

```
p => p.Precio > 100      // dado p, ¿su precio es mayor a 100?
p => p.Nombre           // dado p, dame su nombre
x => x * 2              // dado x, multiplícalo por 2
() => DateTime.Now      // sin parámetros, dame la hora actual
```

No te asustes con la sintaxis: con una semana de práctica la lees como español. Las lambdas aparecen en LINQ, en `RemoveAll`, en eventos y en muchos métodos modernos. Es una de esas sintaxis que al principio se ven raras y al mes ya no puedes vivir sin ellas.

Los operadores esenciales, uno por uno

Vamos con los que usarás el 95 por ciento del tiempo. Supón esta lista de productos para todos los ejemplos:

```
var productos = new List<Producto>
{
    new Producto { Nombre = "Laptop", Precio = 15000, Categoria = "Electrónica",
Existencia = 5 },
    new Producto { Nombre = "Mouse", Precio = 350, Categoria = "Electrónica",
Existencia = 50 },
    new Producto { Nombre = "Teclado", Precio = 800, Categoria = "Electrónica",
Existencia = 30 },
    new Producto { Nombre = "Silla", Precio = 2500, Categoria = "Muebles",
Existencia = 10 },
    new Producto { Nombre = "Mesa", Precio = 4000, Categoria = "Muebles",
Existencia = 0 },
    new Producto { Nombre = "Lámpara", Precio = 600, Categoria = "Muebles",
Existencia = 15 },
};
```

Where: filtrar. Devuelve los elementos que cumplen la condición.

```
var electronica = productos.Where(p => p.Categoria == "Electrónica");
var caros = productos.Where(p => p.Precio > 1000);
var carosConExistencia = productos.Where(p => p.Precio > 1000 && p.Existencia > 0);
```

Select: transformar o proyectar. Devuelve UNA COSA NUEVA por cada elemento. Úsalo para quedarte solo con lo que necesitas:

```
var nombres = productos.Select(p => p.Nombre); // solo los nombres
var etiquetas = productos.Select(p => p.Nombre + " - $" + p.Precio); // texto combinado
var conIVA = productos.Select(p => new Producto
{
    Nombre = p.Nombre,
    Precio = p.Precio * 1.16m, // precio con IVA
    Categoria = p.Categoria,
    Existencia = p.Existencia
});
```

Fíjate: `Select` no modifica la lista original; crea una secuencia nueva con la transformación. Eso es importante: LINQ casi nunca modifica la fuente, produce resultados nuevos. Es más seguro así.

OrderBy y OrderByDescending: ordenar. Y ThenBy para desempatar:

```
var porPrecio = productos.OrderBy(p => p.Precio); // de barato a caro
var porPrecioDesc = productos.OrderByDescending(p => p.Precio); // de caro a barato
var porCategoriaYPrecio = productos
    .OrderBy(p => p.Categoria)
    .ThenBy(p => p.Precio); // primero por categoría, dentro de cada una por precio
```

First y FirstOrDefault: quedarse con el primer elemento que cumpla (o el primero a secas):

```
var primero = productos.First(); // el primero de la lista
var laptop = productos.First(p => p.Nombre == "Laptop"); // el primero que se llame Laptop
```

Aquí viene un detalle que ha tumbado miles de programas: si NINGÚN elemento cumple la condición, `First` lanza una excepción (`InvalidOperationException`: "la secuencia no contiene elementos"). La versión segura es `FirstOrDefault`, que devuelve el valor por defecto (null para objetos, 0 para números) si no hay nada:

```
var buscado = productos.FirstOrDefault(p => p.Nombre == "Impresora");
if (buscado != null)
{
    Console.WriteLine("Encontrado: " + buscado.Nombre);
}
else
{
    Console.WriteLine("No hay impresoras en el catálogo.");
}
```

Regla de oro: si existe la más mínima posibilidad de que no haya resultados, usa `FirstOrDefault` y verifica null. `First` solo cuando estés SEGURO de que hay al menos uno (o cuando quieras que truee a propósito porque "aquí siempre debe haber").

También existen `Single` y `SingleOrDefault`: como First, pero exigen que haya EXACTAMENTE UNO. Si hay dos o más que cumplen, truenan. Úsalos cuando la lógica de tu negocio dice "debe haber uno y solo uno", por ejemplo buscar un usuario por su CURP (que es única).

Count, Any y All: preguntas de sí/no y conteos:

```
int total = productos.Count(); // 6
int caros = productos.Where(p => p.Precio > 1000).Count(); // o mejor:
int caros2 = productos.Count(p => p.Precio > 1000); // Count acepta condición directa

bool haySinExistencia = productos.Any(p => p.Existencia == 0); // true (la mesa)
bool todosCaros = productos.All(p => p.Precio > 100); // true
```

`Any()` sin condición pregunta "¿hay al menos un elemento?" y es la forma elegante de verificar que una lista no está vacía: `if (productos.Any())`. Mucho mejor que `Count() > 0` porque `Any` se detiene en cuanto encuentra el primero, sin contar todo.

Sum, Average, Min, Max: matemáticas directas:

```
decimal inventarioTotal = productos.Sum(p => p.Precio * p.Existencia); // valor del inventario
decimal precioPromedio = productos.Average(p => p.Precio);
decimal masCaro = productos.Max(p => p.Precio);
decimal masBarato = productos.Min(p => p.Precio);
```


Con esto ya puedes hacer reportes: "¿cuánto vale mi inventario?", "¿cuál es el ticket promedio?", "¿qué producto es el más caro?". Todo en una línea por pregunta.

Take y Skip: paginación. `Take(5)` toma los primeros 5; `Skip(10).Take(5)` salta 10 y toma 5 (la página 3 de 5 en 5). Así se hacen las páginas de resultados de cualquier tienda en línea.

ToList y ToArray: materializar. Convierten el resultado en una lista o arreglo concretos. Importante por lo que sigue.

Ejecución diferida: el concepto que casi nadie te explica bien

Esto es sutil pero crucial. Cuando escribes `productos.Where(p => p.Precio > 1000)`, LINQ NO recorre la lista en ese momento. Solo ARMA la pregunta y la guarda. La ejecuta hasta que realmente necesitas los datos: al hacer `foreach`, al llamar `ToList()`, `Count()`, `First()`, etc. Esto se llama ejecución diferida (deferred execution).

¿Por qué importa? Dos razones:

1. Si modificas la lista DESPUÉS de armar la consulta pero ANTES de ejecutarla, el resultado refleja los datos nuevos, no los de cuando armaste la consulta. Esto sorprende a muchos:

```
var consulta = productos.Where(p => p.Precio > 1000); // no se ejecuta aún
productos.Add(new Producto { Nombre = "Servidor", Precio = 50000, Categoria =
"Electrónica", Existencia = 2 });
foreach (var p in consulta) // AHORA se ejecuta... ¡e incluye el servidor!
{
    Console.WriteLine(p.Nombre);
}
```

2. Si vas a recorrer el resultado varias veces, o si la lista original puede cambiar y quieres una "foto" del momento, llama `ToList()` para ejecutar de inmediato y guardar los resultados.

No necesitas dominar esto hoy al 100 por ciento, pero cuando un resultado de LINQ "se comporte raro", lo primero que debes sospechar es la ejecución diferida.

Sintaxis de métodos vs sintaxis de consulta: dos formas de decir lo mismo

Todo lo anterior usa SINTAXIS DE MÉTODOS (encadenar métodos con lambdas). Existe otra forma, la SINTAXIS DE CONSULTA, que se parece al SQL de las bases de datos:

```
// Sintaxis de métodos
var resultado = productos
    .Where(p => p.Categoria == "Electrónica")
    .OrderBy(p => p.Precio)
    .Select(p => p.Nombre);

// Sintaxis de consulta (equivalente)
var resultado2 =
    from p in productos
    where p.Categoria == "Electrónica"
    orderby p.Precio
    select p.Nombre;
```

Las palabras clave: `from` (de dónde), `where` (filtro), `orderby` (orden, con `descending` si quieres al revés), `select` (qué quiero de cada uno), `group by` (agrupar), `join` (combinar dos colecciones).

¿Cuál usar? La industria usa mayormente la sintaxis de MÉTODOS, porque se encadena mejor y casi todo el código profesional está escrito así. Pero DEBES saber leer la sintaxis de consulta porque aparece en código viejo, en tutoriales y en consultas complejas con `join` y `group by`, donde a veces se lee mejor. Mi recomendación: escribe con sintaxis de métodos, lee las dos.

El compilador, por cierto, convierte la sintaxis de consulta en llamadas a métodos. Son lo mismo por dentro; es pura preferencia de lectura.

GroupBy y Join: el siguiente nivel de LINQ

Cuando domines lo anterior, dos operadores te abren la puerta a reportes de verdad: `GroupBy` (agrupar) y `Join` (combinar dos colecciones).

GroupBy agrupa elementos que comparten una característica. Imagina ventas y quieres el total por categoría:

```

var porCategoria = productos
    .GroupBy(p => p.Categoria) // agrupa por categoría
    .Select(g => new
    {
        Categoria = g.Key,           // la categoría del grupo
        Cantidad = g.Count(),        // cuántos productos
        ValorInventario = g.Sum(p => p.Precio * p.Existencia)
    });

foreach (var grupo in porCategoria)
{
    Console.WriteLine(grupo.Categoria + ": " + grupo.Cantidad +
        " productos, inventario de $" + grupo.ValorInventario);
}

```

Cada grupo tiene una `Key` (el valor por el que agrupaste) y contiene los elementos del grupo. Es como separar cartas por palo y luego contar cada montón. El `new { ... }` crea un objeto anónimo: un objeto rápido sin necesidad de definir una clase, perfecto para reportes.

Join combina dos colecciones relacionadas, como en las bases de datos. Supón clientes y pedidos:

```

var clientes = new List<Cliente> { /* ... */ };
var pedidos = new List<Pedido> { /* ... */ }; // cada pedido tiene ClienteId

var reporte = clientes.Join(pedidos,
    c => c.Id,           // clave en clientes
    p => p.ClienteId,    // clave en pedidos
    (c, p) => new        // cómo combinar cada pareja que coincida
    {
        Cliente = c.Nombre,
        Producto = p.Producto,
        Total = p.Cantidad * p.Precio
    });

```

En sintaxis de consulta se lee más natural:

```

var reporte2 =
    from c in clientes
    join p in pedidos on c.Id equals p.ClienteId
    select new { Cliente = c.Nombre, Producto = p.Producto, Total = p.Cantidad *
        p.Precio };

```

Esta es una de las razones por las que vale la pena saber leer la sintaxis de consulta: los `join` complejos se entienden mejor así. No necesitas memorizar Join hoy; solo que sepas que existe cuando tengas que cruzar dos listas.

Errores comunes con LINQ (la sección que te ahorra desveladas)

1. `First` sin elementos: ya lo viste. `FirstOrDefault` + verificación de null es tu amigo. Este es probablemente el error LINQ número uno en producción.
2. Modificar la colección mientras LINQ la recorre: igual que con foreach manual. Si filtras una lista y dentro del ciclo quitas elementos de la lista original, truenas. Solución: materializa primero con `ToList()` o usa `RemoveAll`.
3. Olvidar que `Where`/`Select`/`OrderBy` devuelven una secuencia, no una lista: si necesitas indexar (`resultado[0]`) o pasar una `List<T>` a un método, agrega `.ToList()` al final. Y ojo: cada vez que recorres una consulta diferida, SE VUELVE A EJECUTAR. Si la recorres tres veces, el filtro corre tres veces. Para consultas costosas, `ToList()` una vez y reutiliza la lista.
4. `NullReferenceException` dentro de lambdas: `productos.Where(p => p.Nombre.StartsWith("L"))` truenas si algún producto tiene `Nombre` en null. Defiéndete: `p => p.Nombre != null && p.Nombre.StartsWith("L")`. En C# moderno también puedes usar el operador `?.`, pero la verificación explícita nunca falla.
5. Abusar de LINQ: una consulta de 8 operadores encadenados en una sola línea es difícil de leer y de depurar. Si tu consulta no cabe cómodamente en dos o tres líneas, pártela en pasos con nombres de variables descriptivos:

```
// Difícil de leer:
var x = productos.Where(p => p.Existencia > 0).OrderByDescending(p =>
p.Precio).Take(3).Select(p => p.Nombre).ToList();

// Mejor:
var disponibles = productos.Where(p => p.Existencia > 0);
var masCaros = disponibles.OrderByDescending(p => p.Precio);
var top3Nombres = masCaros.Take(3).Select(p => p.Nombre).ToList();
```

Hace lo mismo, pero cualquier persona (incluido tú dentro de seis meses) entiende cada paso.

Buenas prácticas profesionales para cerrar la explicación

- Elige la colección correcta desde el diseño: ¿orden y duplicados? `IList<T>`. ¿búsqueda por clave única? `IDictionary<TKey, TValue>`. ¿sin duplicados? `IHashSet<T>`. Cambiar de colección a mitad del proyecto duele; pensarlo 5 minutos al inicio ahorra días.
- Programa contra interfaces, no contra clases concretas: recibe `IEnumerable<Producto>` o `IList<Producto>` en tus métodos en vez de `List<Producto>` cuando no necesites nada específico de la lista. Así tu método acepta arreglos, listas y resultados LINQ sin cambios.
- Nombra tus lambdas con variables de una letra del tipo (`p`` para producto, `c`` para cliente): es la convención y se lee bien en consultas cortas. En consultas largas, usa nombres completos.
- No mezcles lógica de negocio compleja dentro de lambdas gigantes: si la condición necesita 10 líneas, sácala a un método con nombre claro y úsalo en el `Where``: `productos.Where(EsElegibleParaDescuento)``. Tu yo del futuro te lo agradecerá.
- Cuidado con el rendimiento en datos grandes: LINQ es cómodo pero cada operador recorre datos. Para millones de registros, un diccionario bien diseñado o una consulta a base de datos le gana a una cadena de 6 operadores LINQ. La comodidad tiene un costo; úsala con conciencia.

Ejemplos paso a paso

Ejemplo 1: Catálogo de productos con reportes LINQ

Vamos a armar un programa completo de consola: una mini tienda con catálogo de productos donde puedes filtrar por categoría, buscar por nombre, ver los más caros y generar un reporte de inventario. Este ejemplo junta todo: clases, `List<T>`, `Dictionary<TKey, TValue>` e interfaces en segundo plano, y LINQ al frente.

Primero la clase `Producto`. Nota que implementa `IComparable<Producto>` para poder ordenarse:

```

using System;
using System.Collections.Generic;
using System.Linq;

// La clase implementa IComparable para que Sort() sepa ordenar por precio
class Producto : IComparable<Producto>
{
    public stringCodigo { get; set; }
    public string Nombre { get; set; }
    public string Categoria { get; set; }
    public decimal Precio { get; set; }
    public int Existencia { get; set; }

    // Regla de comparación: por precio, de menor a mayor
    public int CompareTo(Producto otro)
    {
        if (otro == null) return 1;
        return this.Precio.CompareTo(otro.Precio);
    }

    // Para que al imprimir el producto se vea bonito
    public override string ToString()
    {
        return Codigo + " | " + Nombre + " | " + Categoria +
            " | $" + Precio + " | Existencia: " + Existencia;
    }
}

```

Explicación línea por línea:

- ``using System.Linq;`` es obligatorio para que existan `Where`, `Select`, `OrderBy` y compañía.
- La clase tiene 5 propiedades automáticas: código único, nombre, categoría, precio (decimal, el tipo correcto para dinero) y existencia.
- ``IComparable<Producto>`` es el contrato de "sé compararme con otro producto". El método ``CompareTo`` devuelve negativo si este producto es "menor" (más barato), cero si empatan y positivo si es "mayor". Delegamos en el ``CompareTo`` que decimal ya trae.
- Sobrescribimos ``ToString()`` para que ``Console.WriteLine(producto)`` imprima datos útiles en vez de "Producto" a secas. Truco profesional que usarás siempre.

Ahora el programa principal con el catálogo y los reportes:

```

class Tienda
{

```

```

// El catálogo: diccionario para buscar por código al instante
private Dictionary<string, Producto> catalogo = new Dictionary<string,
Producto>();

public void Agregar(Producto p)
{
    // El código es la clave: no puede repetirse
    if (catalogo.ContainsKey(p.Codigo))
    {
        Console.WriteLine("Ya existe un producto con el código " + p.Codigo);
        return;
    }
    catalogo.Add(p.Codigo, p);
}

// Buscar por código: rapidísimo gracias al diccionario
public Producto BuscarPorCodigo(string codigo)
{
    Producto encontrado;
    if (catalogo.TryGetValue(codigo, out encontrado))
    {
        return encontrado;
    }
    return null; // no existe: devolvemos null en vez de tronar
}

// Todos los productos como lista (para usar LINQ cómodamente)
public List<Producto> Todos()
{
    return catalogo.Values.ToList();
}

// Reporte 1: productos de una categoría, ordenados por precio
public void ReportePorCategoria(string categoria)
{
    var filtrados = Todos()
        .Where(p => p.Categoria == categoria)
        .OrderBy(p => p.Precio)
        .ToList();

    Console.WriteLine("--- " + categoria + " (" + filtrados.Count + " productos)
---");
    foreach (var p in filtrados)
    {
        Console.WriteLine(p);
    }
}

```

```

    }
}

// Reporte 2: los N más caros que sí tienen existencia
public void TopMasCaros(int n)
{
    var top = Todos()
        .Where(p => p.Existencia > 0)
        .OrderByDescending(p => p.Precio)
        .Take(n);

    Console.WriteLine("--- Top " + n + " más caros disponibles ---");
    foreach (var p in top)
    {
        Console.WriteLine(p.Nombre + " - $" + p.Precio);
    }
}

// Reporte 3: valor total del inventario y precio promedio
public void ReporteInventario()
{
    var todos = Todos();
    decimal valorTotal = todos.Sum(p => p.Precio * p.Existencia);
    decimal promedio = todos.Average(p => p.Precio);
    int sinExistencia = todos.Count(p => p.Existencia == 0);

    Console.WriteLine("--- Reporte de inventario ---");
    Console.WriteLine("Productos distintos: " + todos.Count);
    Console.WriteLine("Valor total del inventario: $" + valorTotal);
    Console.WriteLine("Precio promedio: $" + Math.Round(promedio, 2));
    Console.WriteLine("Productos agotados: " + sinExistencia);

    // Alerta de los agotados, usando FirstOrDefault de forma segura
    var agotado = todos.FirstOrDefault(p => p.Existencia == 0);
    if (agotado != null)
    {
        Console.WriteLine("Ejemplo agotado: " + agotado.Nombre + " (¡resurte pronto!)");
    }
}
}

```

Explicación de las decisiones importantes:

- El catálogo es un `Dictionary<string, Producto>` con el código como clave: buscar por código es instantáneo aunque haya 100,000 productos. En `Agregar` verificamos `ContainsKey` antes de agregar para no tronar con clave duplicada.
- `BuscarPorCodigo` usa `TryGetValue`: si no existe, devuelve null en vez de lanzar excepción. El que llama decide qué hacer con el null.
- `Todos()` convierte los valores del diccionario a lista con `ToList()`: una "foto" del momento, ideal para consultas LINQ.
- `ReportePorCategoria` encadena `Where` + `OrderBy` + `ToList`: filtra, ordena de barato a caro y materializa. El `Count` de la lista resultante dice cuántos hay.
- `TopMasCaros` usa `Where` + `OrderByDescending` + `Take`: filtra disponibles, ordena de caro a barato, toma los primeros N. Nota que aquí NO llamamos `ToList` porque solo vamos a recorrerlo una vez con `foreach`; la ejecución diferida trabaja a nuestro favor.
- `ReporteInventario` muestra `Sum` (precio por existencia de cada uno, sumado), `Average` (promedio) y `Count` con condición (agotados). `Math.Round(promedio, 2)` deja el promedio con 2 decimales.
- El uso de `FirstOrDefault` con verificación de null es el patrón seguro: si no hay agotados, no truena, simplemente no muestra la alerta.

Y el Main que lo pone todo a funcionar con un menú:

```
class Program
{
    static void Main()
    {
        Tienda tienda = new Tienda();

        // Cargamos datos de ejemplo
        tienda.Agregar(new Producto {Codigo = "E001", Nombre = "Laptop", Categoria = "Electrónica", Precio = 15000, Existencia = 5 });
        tienda.Agregar(new Producto {Codigo = "E002", Nombre = "Mouse", Categoria = "Electrónica", Precio = 350, Existencia = 50 });
        tienda.Agregar(new Producto {Codigo = "E003", Nombre = "Teclado", Categoria = "Electrónica", Precio = 800, Existencia = 30 });
        tienda.Agregar(new Producto {Codigo = "M001", Nombre = "Silla", Categoria = "Muebles", Precio = 2500, Existencia = 10 });
        tienda.Agregar(new Producto {Codigo = "M002", Nombre = "Mesa", Categoria = "Muebles", Precio = 4000, Existencia = 0 });
        tienda.Agregar(new Producto {Codigo = "M003", Nombre = "Lámpara", Categoria = "Muebles", Precio = 600, Existencia = 15 });
```

```

// Búsqueda por código con manejo de null
Console.Write("Código a buscar: ");
string codigo = Console.ReadLine();
Producto p = tienda.BuscarPorCodigo(codigo);
if (p != null)
{
    Console.WriteLine("Encontrado: " + p);
}
else
{
    Console.WriteLine("No existe un producto con ese código.");
}

Console.WriteLine();
tienda.ReportePorCategoria("Electrónica");
Console.WriteLine();
tienda.TopMasCaros(3);
Console.WriteLine();
tienda.ReporteInventario();

// Búsqueda por texto con LINQ: productos cuyo nombre contenga lo que escribió
el usuario
Console.WriteLine();
Console.Write("Buscar por nombre: ");
string texto = Console.ReadLine();
var coincidencias = tienda.Todos()
    .Where(prod => prod.Nombre.ToLower().Contains(texto.ToLower()))
    .ToList();

if (coincidencias.Any())
{
    Console.WriteLine("Coincidencias:");
    foreach (var c in coincidencias)
    {
        Console.WriteLine(c);
    }
}
else
{
    Console.WriteLine("Sin coincidencias.");
}
}
}

```

La búsqueda por texto convierte todo a minúsculas con `ToLower()` antes de comparar, para que "laptop", "Laptop" y "LAPTOP" coincidan. Es el truco estándar para búsquedas insensibles a mayúsculas. Y `Any()` verifica elegantemente si hubo resultados.

Ejemplo 2: Agenda de contactos con Dictionary

Ahora un ejemplo enfocado en `Dictionary<TKey, TValue>`: una agenda telefónica completa con agregar, buscar, actualizar y borrar. Es el programa perfecto para entender claves, `TryGetValue` y recorridos.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Contacto
{
    public string Nombre { get; set; }
    public string Telefono { get; set; }
    public string Email { get; set; }

    public override string ToString()
    {
        return Nombre + " | Tel: " + Telefono + " | Email: " + Email;
    }
}

class Agenda
{
    // Clave: nombre del contacto (único). Valor: el contacto completo.
    private Dictionary<string, Contacto> contactos = new Dictionary<string,
Contacto>();

    public void Agregar(string nombre, string telefono, string email)
    {
        string clave = nombre.ToLower(); // clave normalizada: "Ana" y "ANA" son la
misma persona

        if (contactos.ContainsKey(clave))
        {
            Console.WriteLine("'" + nombre + "' ya existe. Usa la opción
actualizar.");
            return;
        }
    }
}
```

```

        contactos.Add(clave, new Contacto { Nombre = nombre, Telefono = telefono,
Email = email });
        Console.WriteLine("Contacto agregado.");
    }

    public void Buscar(string nombre)
    {
        string clave = nombre.ToLower();
        Contacto c;

        // TryGetValue: la forma profesional de buscar sin riesgo de excepción
        if (contactos.TryGetValue(clave, out c))
        {
            Console.WriteLine("Encontrado: " + c);
        }
        else
        {
            Console.WriteLine("No hay ningún contacto llamado '" + nombre + "'.");
        }
    }

    public void ActualizarTelefono(string nombre, string nuevoTelefono)
    {
        string clave = nombre.ToLower();
        Contacto c;

        if (contactos.TryGetValue(clave, out c))
        {
            c.Telefono = nuevoTelefono; // el objeto en el diccionario se actualiza
directamente
            Console.WriteLine("Teléfono actualizado.");
        }
        else
        {
            Console.WriteLine("No existe '" + nombre + "'.");
        }
    }

    public void Eliminar(string nombre)
    {
        string clave = nombre.ToLower();

        // Remove devuelve true si sí lo quitó, false si no existía
        if (contactos.Remove(clave))
        {

```

```

        Console.WriteLine("Contacto eliminado.");
    }
    else
    {
        Console.WriteLine("No existe '" + nombre + "'.");
    }
}

public void MostrarTodos()
{
    if (!contactos.Any())
    {
        Console.WriteLine("La agenda está vacía.");
        return;
    }

    // OrderBy sobre los valores del diccionario: LINQ trabaja sobre cualquier
IEnumerable
    var ordenados = contactos.Values.OrderBy(c => c.Nombre);

    Console.WriteLine("--- Contactos (" + contactos.Count + ") ---");
    foreach (var c in ordenados)
    {
        Console.WriteLine(c);
    }
}

// Búsqueda parcial: "todos los que empiecen con..."
public void BuscarPorPrefijo(string prefijo)
{
    var resultados = contactos.Values
        .Where(c => c.Nombre.ToLower().StartsWith(prefijo.ToLower()))
        .OrderBy(c => c.Nombre)
        .ToList();

    Console.WriteLine("Contactos que empiezan con '" + prefijo + "': " +
resultados.Count);
    foreach (var c in resultados)
    {
        Console.WriteLine(c);
    }
}
}

```

Explicación de los puntos finos:

- La clave del diccionario es el nombre en minúsculas (`nombre.ToLower()`). Así "Ana", "ANA" y "ana" son la misma clave y no terminas con tres Anas distintas. Normalizar las claves es una práctica profesional: decide UN formato y úsalo siempre al guardar y al buscar.
- ``Agregar`` verifica ``ContainsKey`` antes: evita la excepción por clave duplicada y además da un mensaje útil ("ya existe, usa actualizar"). Eso es diseñar pensando en el usuario.
- ``TryGetValue`` aparece en `Buscar` y `ActualizarTelefono`: un solo acceso al diccionario que busca y obtiene. La alternativa ingenua (``ContainsKey`` + indexador) accede dos veces; ``TryGetValue`` lo hace una sola vez. Más rápido y más elegante.
- En ``ActualizarTelefono`` modificamos ``c.Telefono`` directamente: como ``Contacto`` es una clase (tipo por referencia), la variable ``c`` apunta al mismo objeto que está guardado en el diccionario. Cambiarlo ahí lo cambia en la agenda. No hay que "volver a guardarlo".
- ``Remove`` devuelve bool: úsalo para saber si realmente se eliminó algo, en vez de preguntar antes con ``ContainsKey``.
- ``MostrarTodos`` ordena con LINQ sobre ``contactos.Values``: demuestra que LINQ no es solo para listas; trabaja sobre cualquier colección enumerable, incluidos los valores de un diccionario.
- ``BuscarPorPrefijo`` combina ``StartsWith`` con LINQ: es la base de los autocompletados ("escribe y te sugiero"). Con un diccionario de miles de contactos, esto sigue siendo rápido porque solo recorre valores una vez.

El Main con menú interactivo:

```
class Program
{
    static void Main()
    {
        Agenda agenda = new Agenda();
        string opcion = "";

        while (opcion != "6")
        {
            Console.WriteLine();
            Console.WriteLine("=== AGENDA ===");
            Console.WriteLine("1. Agregar contacto");
            Console.WriteLine("2. Buscar contacto");
            Console.WriteLine("3. Actualizar teléfono");
            Console.WriteLine("4. Eliminar contacto");
            Console.WriteLine("5. Mostrar todos");
            Console.WriteLine("6. Salir");
        }
    }
}
```

```

        Console.Write("Opción: ");
        opcion = Console.ReadLine();

        if (opcion == "1")
        {
            Console.Write("Nombre: ");
            string n = Console.ReadLine();
            Console.Write("Teléfono: ");
            string t = Console.ReadLine();
            Console.Write("Email: ");
            string e = Console.ReadLine();
            agenda.Agregar(n, t, e);
        }
        else if (opcion == "2")
        {
            Console.Write("Nombre a buscar: ");
            agenda.Buscar(Console.ReadLine());
        }
        else if (opcion == "3")
        {
            Console.Write("Nombre: ");
            string n = Console.ReadLine();
            Console.Write("Nuevo teléfono: ");
            string t = Console.ReadLine();
            agenda.ActualizarTelefono(n, t);
        }
        else if (opcion == "4")
        {
            Console.Write("Nombre a eliminar: ");
            agenda.Eliminar(Console.ReadLine());
        }
        else if (opcion == "5")
        {
            agenda.MostrarTodos();
        }
    }

    Console.WriteLine("¡Nos vemos!");
}
}

```

Este menú es el esqueleto de cientos de programas reales: un ciclo que muestra opciones, lee la elección y delega en métodos. La agenda por dentro usa el diccionario; el usuario del menú ni se entera. Esa separación (interfaz de usuario vs almacenamiento) es exactamente la forma de pensar que las interfaces promueven: el menú no depende de CÓMO se guardan los contactos, solo de QUÉ puede hacer la agenda.

Ejemplo 3: Múltiples interfaces en acción (sistema de notificaciones)

Un ejemplo corto pero muy ilustrativo de por qué las interfaces valen oro. Un sistema que envía notificaciones por varios canales:

```
using System;
using System.Collections.Generic;

// Contrato: todo canal de notificación sabe enviar un mensaje
interface INotificador
{
    void Enviar(string destinatario, string mensaje);
    string NombreCanal { get; }
}

class NotificadorEmail : INotificador
{
    public string NombreCanal { get { return "Email"; } }

    public void Enviar(string destinatario, string mensaje)
    {
        Console.WriteLine("[Email a " + destinatario + "]: " + mensaje);
    }
}

class NotificadorSMS : INotificador
{
    public string NombreCanal { get { return "SMS"; } }

    public void Enviar(string destinatario, string mensaje)
    {
        Console.WriteLine("[SMS a " + destinatario + "]: " + mensaje);
    }
}

class NotificadorPush : INotificador
{
    public string NombreCanal { get { return "Push"; } }
```



```

        public void Enviar(string destinatario, string mensaje)
        {
            Console.WriteLine("[Push a " + destinatario + "]: " + mensaje);
        }
    }

class SistemaAlertas
{
    // Lista de CONTRATOS, no de clases concretas
    private List<INotificador> canales = new List<INotificador>();

    public void RegistrarCanal(INotificador canal)
    {
        canales.Add(canal);
    }

    // Envía por TODOS los canales registrados, sin saber cuáles son
    public void AlertarATodos(string mensaje)
    {
        foreach (INotificador canal in canales)
        {
            canal.Enviar("todos", mensaje);
        }
    }
}

class Program
{
    static void Main()
    {
        SistemaAlertas sistema = new SistemaAlertas();
        sistema.RegistrarCanal(new NotificadorEmail());
        sistema.RegistrarCanal(new NotificadorSMS());
        sistema.RegistrarCanal(new NotificadorPush());

        sistema.AlertarATodos("El servidor necesita mantenimiento.");

        // Mañana agregan WhatsApp: solo creas la clase, el sistema no cambia
    }
}

```

Fíjate en lo poderoso: `SistemaAlertas` guarda `List<INotificador>`. No sabe ni le importa si el canal es email, SMS o algo que aún no existe. Si mañana el negocio pide notificaciones por WhatsApp, creas `NotificadorWhatsApp : INotificador`, lo registras, y `AlertarATodos` funciona

sin tocar una sola línea del sistema. Ese es el principio abierto/cerrado en acción: abierto a extensión (nuevos canales), cerrado a modificación (el sistema no cambia). Cuando en una entrevista te pregunten "¿para qué sirven las interfaces?", cuenta esta historia.

Práctica

Resuelve primero sin mirar las soluciones. Escribe el código, ejecútalo, y si truena, lee el mensaje de error con calma: los mensajes de C# suelen decirte exactamente qué pasó. Después compara con las soluciones: no importa si tu código es distinto, importa que funcione y que entiendas por qué.

1. Calificaciones con `List<T>`: crea una lista con 8 calificaciones (`double`). Calcula con LINQ: el promedio, la más alta, la más baja y cuántas son aprobatorias (mayor o igual a 6). Muestra los resultados.
2. Filtro de palabras: dada una lista de palabras, usa `Where` para quedarte con las que tengan más de 5 letras, ordénalas alfabéticamente con `OrderBy` y muéstralas en mayúsculas usando `Select`. Todo encadenado en una sola consulta.
3. Mini diccionario español-inglés: crea un `Dictionary<string, string>` con al menos 6 traducciones. Pide al usuario una palabra en español y muestra su traducción usando `TryGetValue`. Si no existe, muestra un mensaje amable en vez de tronar.
4. Múltiples interfaces: define las interfaces `IElectrico` (método `CargarBateria`) e `IAutonomo` (método `ConducirSolo`). Crea la clase `AutoTesla` que implemente AMBAS, con mensajes distintos en cada método. En el `Main`, guarda el objeto en una variable de tipo `IElectrico` y en otra de tipo `IAutonomo`, y llama los métodos desde cada una.
5. Reporte de ventas con LINQ: crea una lista de 6 ventas (cada venta: producto, cantidad, precio unitario). Calcula: el ingreso total (`Sum` de cantidad por precio), la venta con mayor ingreso (`OrderByDescending + First`), el ingreso promedio por venta (`Average`) y cuántas ventas superan los \$1000 de ingreso (`Count` con condición).
6. Agenda mejorada: extiende el ejemplo de la agenda para que también guarde la ciudad del contacto. Agrega un método que, usando LINQ sobre los valores del diccionario, muestre cuántos contactos hay por ciudad (pista: agrupa mentalmente filtrando por cada ciudad, o usa un diccionario auxiliar ciudad -> conteo).
7. Sintaxis de consulta: toma el ejemplo del catálogo de productos y reescribe el reporte por categoría usando sintaxis de consulta (`from`, `where`, `orderby`, `select`) en vez de sintaxis de métodos. Verifica que el resultado sea idéntico.

8. Quitar sin tronar: crea una lista de números del 1 al 20. Elimina todos los múltiplos de 3 SIN usar foreach con Remove adentro (usa RemoveAll o un for hacia atrás). Muestra la lista antes y después, y usa Count para verificar cuántos quedaron.

Soluciones

Solución 1: calificaciones con LINQ. La clave es que Average, Max, Min y Count aceptan la lista directo, y Count acepta una condición lambda:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        List<double> calificaciones = new List<double> { 8.5, 5.9, 7.0, 9.2, 6.0, 4.8, 10.0, 7.7 };

        double promedio = calificaciones.Average();
        double maxima = calificaciones.Max();
        double minima = calificaciones.Min();
        int aprobadas = calificaciones.Count(c => c >= 6);

        Console.WriteLine("Promedio: " + Math.Round(promedio, 2));
        Console.WriteLine("Más alta: " + maxima);
        Console.WriteLine("Más baja: " + minima);
        Console.WriteLine("Aprobatorias: " + aprobadas + " de " + calificaciones.Count());
    }
}
```

Nota que `Count(c => c >= 6)` cuenta directamente con condición, sin necesidad de un Where previo. Y `Math.Round(promedio, 2)` evita un promedio con 15 decimales.

Solución 2: filtro encadenado. Un solo flujo Where -> OrderBy -> Select, cada paso con una responsabilidad:

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        List<string> palabras = new List<string>
        {
            "sol", "computadora", "luz", "programación", "mar", "teclado", "nube",
            "algoritmo"
        };

        var resultado = palabras
            .Where(p => p.Length > 5)    // filtra: más de 5 letras
            .OrderBy(p => p)              // ordena alfabéticamente
            .Select(p => p.ToUpper());   // transforma a mayúsculas

        foreach (string s in resultado)
        {
            Console.WriteLine(s);
        }
    }
}

```

El orden de los operadores importa: filtrar primero reduce los elementos que los pasos siguientes procesan. En listas pequeñas no se nota, en grandes sí.

Solución 3: diccionario con TryGetValue. El patrón seguro de búsqueda:

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<string, string> traductor = new Dictionary<string, string>();
        traductor.Add("casa", "house");
        traductor.Add("perro", "dog");
        traductor.Add("libro", "book");
        traductor.Add("agua", "water");
        traductor.Add("sol", "sun");
        traductor.Add("computadora", "computer");

        Console.Write("Palabra en español: ");
        string palabra = Console.ReadLine().ToLower();

        string traduccion;
        if (traductor.TryGetValue(palabra, out traduccion))
        {
            Console.WriteLine(palabra + " en inglés es: " + traduccion);
        }
        else
        {
            Console.WriteLine("Lo siento, no conozco la traducción de '" + palabra
+ "'.");
        }
    }
}

```

Convertimos la entrada a minúsculas para que "Casa" y "casa" encuentren lo mismo. Si hubiéramos usado `traductor[palabra]` directamente con una palabra inexistente, el programa habría tronado con `KeyNotFoundException`.

Solución 4: múltiples interfaces. Una clase, dos contratos:

```

using System;

interface IElectrico
{
    void CargarBateria();
}

interface IAutonomo
{
    void ConducirSolo();
}

class AutoTesla : IElectrico, IAutonomo
{
    public void CargarBateria()
    {
        Console.WriteLine("Conectando al supercargador... batería al 100%.");
    }

    public void ConducirSolo()
    {
        Console.WriteLine("Piloto automático activado. Tú relájate.");
    }
}

class Program
{
    static void Main()
    {
        AutoTesla miAuto = new AutoTesla();

        // El mismo objeto visto a través de dos contratos distintos
        IElectrico comoElectrico = miAuto;
        comoElectrico.CargarBateria();

        IAutonomo comoAutonomo = miAuto;
        comoAutonomo.ConducirSolo();
    }
}

```

Desde la variable `IElectrico` solo se ve `CargarBateria`; desde `IAutonomo` solo `ConducirSolo`. Es el mismo objeto, pero cada contexto usa solo la capacidad que le importa. Así funcionan los sistemas reales: cada parte del programa ve solo lo que necesita.

Solución 5: reporte de ventas. Aquí brilla LINQ para cálculos de negocio:

```

using System;
using System.Collections.Generic;
using System.Linq;

class Venta
{
    public string Producto { get; set; }
    public int Cantidad { get; set; }
    public decimal PrecioUnitario { get; set; }

    // Propiedad calculada: el ingreso de esta venta
    public decimal Ingreso { get { return Cantidad * PrecioUnitario; } }
}

class Program
{
    static void Main()
    {
        List<Venta> ventas = new List<Venta>
        {
            new Venta { Producto = "Laptop", Cantidad = 2, PrecioUnitario = 15000
},
            new Venta { Producto = "Mouse", Cantidad = 20, PrecioUnitario = 350 },
            new Venta { Producto = "Teclado", Cantidad = 10, PrecioUnitario = 800
},
            new Venta { Producto = "Monitor", Cantidad = 5, PrecioUnitario = 4500
},
            new Venta { Producto = "Silla", Cantidad = 3, PrecioUnitario = 2500 },
            new Venta { Producto = "USB", Cantidad = 50, PrecioUnitario = 150 },
        };

        decimal ingresoTotal = ventas.Sum(v => v.Ingreso);
        Venta mejorVenta = ventas.OrderByDescending(v => v.Ingreso).First();
        decimal promedio = ventas.Average(v => v.Ingreso);
        int grandes = ventas.Count(v => v.Ingreso > 1000);

        Console.WriteLine("Ingreso total: $" + ingresoTotal);
        Console.WriteLine("Mejor venta: " + mejorVenta.Producto + " (" + $" +
mejorVenta.Ingreso + ")");
        Console.WriteLine("Ingreso promedio por venta: $" + Math.Round(promedio,
2));
        Console.WriteLine("Ventas mayores a $1000: " + grandes);
    }
}

```


El truco elegante es la propiedad calculada ``Ingreso``: la lógica de "cantidad por precio" vive en la clase `Venta`, y LINQ solo la usa. Si mañana el ingreso incluye impuestos, cambias UN lugar. Además, ``First()`` aquí es seguro porque la lista nunca está vacía (la creamos con datos); en datos que vienen de fuera usarías ``FirstOrDefault``.

Solución 6: agenda con conteo por ciudad. Usamos un diccionario auxiliar para contar:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Contacto
{
    public string Nombre { get; set; }
    public string Telefono { get; set; }
    public string Ciudad { get; set; }
}

class Program
{
    static void Main()
    {
        Dictionary<string, Contacto> agenda = new Dictionary<string, Contacto>();
        agenda.Add("ana", new Contacto { Nombre = "Ana", Telefono = "555-0001", Ciudad = "CDMX" });
        agenda.Add("beto", new Contacto { Nombre = "Beto", Telefono = "555-0002", Ciudad = "Guadalajara" });
        agenda.Add("carla", new Contacto { Nombre = "Carla", Telefono = "555-0003", Ciudad = "CDMX" });
        agenda.Add("david", new Contacto { Nombre = "David", Telefono = "555-0004", Ciudad = "Monterrey" });
        agenda.Add("elena", new Contacto { Nombre = "Elena", Telefono = "555-0005", Ciudad = "CDMX" });

        // Diccionario auxiliar: ciudad -> cuántos contactos
        Dictionary<string, int> porCiudad = new Dictionary<string, int>();

        foreach (Contacto c in agenda.Values)
        {
            if (porCiudad.ContainsKey(c.Ciudad))
            {
                porCiudad[c.Ciudad]++; // ya existe: suma uno
            }
            else
            {

```

```

        porCiudad[c.Ciudad] = 1; // primera vez: empieza en 1
    }
}

Console.WriteLine("--- Contactos por ciudad ---");
foreach (var par in porCiudad.OrderBy(p => p.Key))
{
    Console.WriteLine(par.Key + ": " + par.Value);
}
}
}

```

Este patrón "diccionario para contar" aparece en todas partes: contar palabras en un texto, votos por candidato, errores por tipo. Apréndetelo: si la clave existe incrementas, si no la creas en 1. (En LINQ avanzado existe `GroupBy` que hace esto en una línea, pero entender el patrón manual primero te hace mejor programador.)

Solución 7: sintaxis de consulta. El mismo reporte, otra sintaxis:

```

// Sintaxis de métodos (original):
var filtrados = tienda.Todos()
    .Where(p => p.Categoria == categoria)
    .OrderBy(p => p.Precio)
    .Select(p => p);

// Sintaxis de consulta (equivalente):
var filtrados2 =
    from p in tienda.Todos()
    where p.Categoria == categoria
    orderby p.Precio
    select p;

```

Ambas producen exactamente lo mismo. La sintaxis de consulta se lee como una frase: "de p en todos, donde la categoría sea tal, ordenado por precio, selecciona p". Practica leerla porque la encontrarás en código existente, aunque para escribir nuevo prefieras la de métodos.

Solución 8: quitar múltiplos de 3 sin tronar. Dos formas correctas:

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        List<int> numeros = new List<int>();
        for (int i = 1; i <= 20; i++)
        {
            numeros.Add(i);
        }

        Console.WriteLine("Antes (" + numeros.Count + "): " + string.Join(", ",
numeros));

        // Forma 1: RemoveAll con lambda (la más elegante)
        numeros.RemoveAll(n => n % 3 == 0);

        Console.WriteLine("Después (" + numeros.Count + "): " + string.Join(", ",
numeros));

        // Forma 2 (alternativa): for hacia atrás con RemoveAt
        // for (int i = numeros.Count - 1; i >= 0; i--)
        // {
        //     if (numeros[i] % 3 == 0) numeros.RemoveAt(i);
        // }
    }
}

```

`RemoveAll` recibe una condición y quita todos los que la cumplan, de forma segura y en una línea. `string.Join(", ", numeros)` convierte la lista en texto separado por comas para mostrarla bonito. Quedan 14 números (se quitaron 3, 6, 9, 12, 15 y 18).

Resumen de la unidad

- Una interfaz es un contrato: declara QUÉ debe poder hacer una clase, sin decir CÓMO. Se declara con `interface`, sus miembros no llevan código, y por convención el nombre empieza con I.

- Una clase implementa una interfaz con `:`` y está obligada a incluir TODOS sus miembros. No puedes crear objetos de una interfaz, pero sí variables de ese tipo que apunten a objetos de clases que la implementen.
- Una clase hereda de una sola base pero puede implementar muchas interfaces. Úsalo para dar capacidades a clases no relacionadas (el pato que vuela y nada, el auto que es eléctrico y autónomo).
- Interfaz vs clase abstracta: interfaz para CAPACIDADES compartidas entre cosas distintas ("puede volar"); clase abstracta para FAMILIAS con código y estado en común ("es un animal").
- ``List<T>`` es tu colección por defecto: secuencia ordenada que crece sola, permite duplicados. Domina Add, Insert, Remove, RemoveAt, Clear, Contains, IndexOf, Count, Sort y los recorridos con foreach.
- ``Dictionary<TKey, TValue>`` guarda pares clave-valor con búsqueda instantánea. Las claves no se repiten. Nunca leas con el indexador sin verificar: usa ``ContainsKey`` o mejor ``TryGetValue``. Recorre con ``foreach`` sobre el diccionario (pares), ``.Keys`` o ``.Values``.
- Elige la colección por su uso: List para secuencias ordenadas, Dictionary para búsquedas por clave única, HashSet para evitar duplicados, Queue para filas (primero en entrar, primero en salir), Stack para pilas (último en entrar, primero en salir).
- NUNCA modifiques una lista dentro del foreach que la recorre: lanza InvalidOperationException. Usa ``ToList()`` para recorrer una copia, un ``for`` hacia atrás, o ``RemoveAll`` con lambda.
- LINQ te deja consultar colecciones describiendo QUÉ quieres: Where filtra, Select transforma, OrderBy/OrderByDescending ordenan (ThenBy desempata), Take/Skip paginan, ToList materializa.
- First devuelve el primero o TRUENA si no hay; FirstOrDefault devuelve null (o el valor por defecto) si no hay: úsalo cuando el vacío sea posible y verifica el resultado. Single exige exactamente uno.
- Count, Any, All, Sum, Average, Min y Max responden preguntas sobre los datos en una línea. ``Any()`` es la forma correcta de preguntar si una colección tiene elementos.
- Las expresiones lambda (``p => p.Precio > 100``) son funciones pequeñas en línea: la flecha se lee "dado que devuelve" o "tal que".
- LINQ tiene ejecución diferida: la consulta se arma al escribirla y se ejecuta al recorrerla o materializarla. Si necesitas una foto del momento o vas a reutilizar el resultado, llama ``ToList()``.

- Existen dos sintaxis equivalentes: la de métodos (encadenar con lambdas, la más usada en la industria) y la de consulta (from/where/orderby/select, estilo SQL). Escribe con la que prefieras, pero sabe leer las dos.
- Buenas prácticas: programa contra interfaces (recibe `IEnumerable<T>` en vez de `List<T>` cuando puedas), mantén las consultas LINQ legibles partiéndolas en pasos, no metas lógica compleja en lambdas gigantes, y elige la colección correcta desde el diseño porque cambiarla después duele.

UNIDAD 8: Excepciones, archivos y cadenas

Hasta ahora tus programas viven en una burbuja perfecta: el usuario siempre escribe lo que esperas, los archivos siempre existen, la memoria nunca falla y las palabras siempre vienen bien escritas. En el mundo real nada de eso es cierto. El usuario escribe "hola" donde pediste un número, el archivo que buscas lo borraron ayer, el disco se llenó a la mitad de guardar, y la cadena que llegó de internet viene vacía. Esta unidad es sobre programar para el mundo real: qué hacer cuando las cosas salen mal, cómo guardar datos que sobrevivan al apagar la computadora, y cómo manipular texto como un profesional.

Vas a ver tres temas. Primero, las excepciones: el mecanismo de C# para manejar errores en tiempo de ejecución sin que tu programa se desplome de forma fea. Aprenderás try/catch/finally, los tipos de excepción más comunes (FormatException, NullReferenceException, DivideByZeroException), cómo lanzar las tuyas con throw, cómo crear tus propios tipos de excepción, y lo más importante: el criterio para decidir cuándo atrapar un error y cuándo dejarlo pasar. Porque atrapar todo "por si acaso" es tan malo como no atrapar nada.

Segundo, los archivos: cómo leer y escribir datos en disco. Con File.WriteAllText y File.ReadAllText guardarás información en dos líneas; con StreamReader y StreamWriter manejarás archivos grandes o escrituras por partes; y con using te asegurarás de liberar los recursos siempre, incluso si algo truena a la mitad. También verás rutas: cómo armarlas bien con Path.Combine para que tu programa funcione en cualquier computadora, no solo en la tuya.

Tercero, las cadenas avanzadas: el texto es el dato más común en programación (nombres, direcciones, mensajes, archivos de configuración, páginas web), así que vale la pena dominarlo. Verás la interpolación \$"", que hace legible la construcción de mensajes; StringBuilder para armar textos grandes sin matar el rendimiento; y los métodos que usarás a diario: Split, Join, Contains, StartsWith, Replace, Substring, Trim. Además, el formato de números y fechas: cómo mostrar \$1,234.56 en vez de 1234.56, y fechas legibles en vez de códigos raros.

¿Por qué importa en la vida real? Porque un programa que se cae con un mensaje críptico en inglés cuando el usuario escribe mal un dato es un programa que nadie quiere usar. Las empresas pierden clientes y dinero por programas frágiles. Un sistema de punto de venta que pierde la venta porque "tronó al guardar el ticket" es un desastre; uno que atrapa el error, guarda un respaldo y le dice al cajero "hubo un problema, tu venta está a salvo" es un sistema profesional. La diferencia entre ambos no es más código: es manejo de excepciones y

archivos bien hechos. Y las cadenas... todo programa serio procesa texto: reportes, búsquedas, validaciones, formatos. Dominar las cadenas es dominar la mitad del trabajo diario.

Al terminar esta unidad podrás: envolver código riesgoso en try/catch/finally y reaccionar a cada tipo de error de forma distinta, lanzar excepciones con throw cuando tus propios métodos detecten algo inválido, crear tipos de excepción propios para tu negocio, guardar y leer archivos de texto de forma segura con using, armar rutas portables con Path.Combine, construir mensajes con interpolación, procesar texto con Split/Join/Contains/StartsWith, usar StringBuilder cuando el rendimiento importe, y dar formato profesional a números y fechas. Y lo demostrarás con dos programas completos: un bloc de notas que guarda y lee tus notas en disco, y un procesador de texto que cuenta palabras, busca y reemplaza.

Un consejo de colega: esta unidad tiene menos "magia" que la anterior y más "oficio". No hay atajos: la única forma de aprender manejo de errores es provocando errores a propósito. Borra el archivo antes de leerlo, escribe letras donde pide números, desconecta... bueno, no desconectes nada, pero sí fuerza cada excepción que veas y mira cómo se comporta tu programa. El programador que ya vio todos los errores es el que no se asusta cuando aparecen en producción. Vamos.

Explicación detallada

Clase 22: Excepciones, cuando las cosas salen mal (y van a salir mal)

Qué es una excepción y en qué se diferencia de un error de compilación

Hay dos familias de errores en programación, y confundirlas es el primer tropiezo:

- Errores de COMPILACIÓN: el código está mal escrito (te faltó un punto y coma, usaste una variable que no existe). El programa ni siquiera arranca. El compilador te los señala todos antes de ejecutar. Son molestos pero inofensivos: nadie los ve más que tú.
- Errores de EJECUCIÓN (excepciones): el código está bien escrito, pero algo sale mal MIENTRAS el programa corre: el usuario escribió "abc" donde iba un número, el archivo no existe, se dividió entre cero, la red se cayó. El programa ya está en manos del usuario cuando ocurren. Estos son los peligrosos.

Una excepción es el objeto que C# crea cuando ocurre un error de ejecución. Piensa en ella como una "alarma con información": no solo dice "algo salió mal", trae datos: qué tipo de error fue, en qué línea ocurrió, qué método la provocó y la cadena de llamadas que llevó hasta ahí.

Si nadie la "atiende", la alarma sube hasta el sistema operativo y tu programa se cierra de golpe con un mensaje técnico que asusta al usuario. Si la atiendes con try/catch, tú decides qué hacer: reintentar, usar un valor por defecto, guardar un respaldo o mostrar un mensaje amable.

Analogía: es como el sistema de alarmas de un edificio. Sin alarma, un incendio se descubre cuando ya se quemó todo. Con alarma, suena, dice en qué piso es, y el personal decide: evacuar, usar extinguidores o llamar a los bomberos. El try/catch es tu personal entrenado.

try/catch/finally: la estructura completa

```
try
{
    // Código RIESGOSO: aquí puede tronar algo
    Console.Write("Escribe un número: ");
    int n = int.Parse(Console.ReadLine());
    int resultado = 100 / n;
    Console.WriteLine("100 / " + n + " = " + resultado);
}
catch (FormatException)
{
    // Se ejecuta si el texto no era un número
    Console.WriteLine("Eso no es un número válido.");
}
catch (DivideByZeroException)
{
    // Se ejecuta si n fue cero
    Console.WriteLine("No se puede dividir entre cero.");
}
catch (Exception ex)
{
    // Red de seguridad: cualquier otra excepción
    Console.WriteLine("Error inesperado: " + ex.Message);
}
finally
{
    // Se ejecuta SIEMPRE: haya error o no, haya catch o no
    Console.WriteLine("Gracias por usar el programa.");
}
```

Cómo funciona, paso a paso:

1. Se ejecuta el bloque ``try``. Si todo sale bien, se SALTA todos los `catch` y se ejecuta el ``finally``. Fin.
2. Si dentro del `try` ocurre una excepción, la ejecución del `try` se DETIENE en seco en esa línea (lo que sigue no se ejecuta) y C# busca un ``catch`` que corresponda al tipo de excepción.
3. Los `catch` se revisan EN ORDEN, de arriba a abajo. El primero cuyo tipo coincida (o sea un tipo base del tipo lanzado) atiende la excepción. Por eso los `catch` específicos van ANTES que el genérico ``Exception``: si pones ``catch (Exception)`` primero, atraparé todo y los específicos de abajo jamás se ejecutarán (el compilador incluso te marca error).
4. El ``finally`` se ejecuta SIEMPRE: si hubo excepción o no, si se atendió o no. Es el lugar para limpieza: cerrar archivos, liberar conexiones, soltar recursos. En la clase 23 verás que ``using`` es básicamente un `finally` automático.

Puedes tener `try/catch` sin `finally`, y `try/finally` sin `catch` (cuando solo quieres garantizar limpieza y dejar que la excepción siga subiendo). Lo que NO puedes es tener `catch` o `finally` sin `try`.

Las excepciones que verás toda tu vida (memorízalas)

Estas son las que aparecen una y otra vez. Conocerlas por nombre te permite reaccionar distinto a cada una:

- ``FormatException``: "el texto no tiene el formato esperado". Clásica de ``int.Parse("hola")`` o ``decimal.Parse("12.34.56")``. Aparece cada vez que conviertes entrada del usuario sin validar.
- ``NullReferenceException``: "intentaste usar algo que es null". La más famosa de .NET. ``string nombre = null; int largo = nombre.Length;`` -> truena. Cada vez que veas esta excepción, pregunta: "¿qué cosa aquí podía venir en null y no lo verifiqué?"
- ``DivideByZeroException``: división entera entre cero. Ojo: con ``double`` no truena (da infinito); con ``int`` sí.
- ``IndexOutOfRangeException``: índice fuera del arreglo o la lista (``arreglo[10]`` cuando solo hay 5). Hermana del error de índices que viste en la unidad 7.
- ``KeyNotFoundException``: leer una clave inexistente en un diccionario. Ya la conoces: por eso usas `TryGetValue`.
- ``InvalidOperationException``: "la operación no es válida en el estado actual". La de modificar una lista dentro de un `foreach`, o llamar ``First()`` en una secuencia vacía.

- ``ArgumentException`` (y sus hijas ``ArgumentNullException``, ``ArgumentOutOfRangeException``): "me pasaste un argumento inválido". La lanzan los métodos cuando reciben parámetros incorrectos, y TÚ deberías lanzarla cuando validas parámetros en tus propios métodos.
- ``IOException`` (y su hija ``FileNotFoundException``): problemas de archivos y disco: el archivo no existe, el disco está lleno, otro programa lo tiene bloqueado.
- ``OutOfMemoryException``: te quedaste sin memoria. Rara en programas normales; si la ves, algo está muy mal (como concatenar strings en un ciclo infinito... ya llegaremos a eso).

Cada excepción trae propiedades útiles:

- ``Message``: descripción del error en texto. Es lo que muestras al usuario (a veces) o guardas en el log (siempre).
- ``StackTrace``: la "pila de llamadas": qué métodos se llamaron, en qué orden, hasta llegar al error, con números de línea. Es tu mapa del tesoro para depurar.
- ``InnerException``: cuando un catch lanza OTRA excepción, la original se guarda aquí. Así no pierdes la causa raíz.
- ``GetType().Name``: el nombre del tipo de excepción, útil para logs.

throw: lanzar tus propias excepciones

No solo puedes ATRAPAR excepciones: también puedes LANZARLAS. ¿Para qué? Para que tus métodos digan "esto que me pides no lo puedo hacer" de forma estándar, en vez de devolver códigos de error raros (-1, null, false) que el que llama puede ignorar sin darse cuenta.

```

class CuentaBancaria
{
    public decimal Saldo { get; private set; }

    public void Retirar(decimal monto)
    {
        if (monto <= 0)
        {
            throw new ArgumentException("El monto debe ser mayor a cero.");
        }

        if (monto > Saldo)
        {
            throw new InvalidOperationException("Fondos insuficientes para el
retiro.");
        }

        Saldo -= monto;
    }
}

```

Fíjate en la filosofía: el método VALIDA sus entradas y su estado, y si algo está mal, lanza una excepción con un mensaje claro en vez de hacer algo incorrecto en silencio. El que llama al método está obligado a lidiar con eso (o el programa truena visiblemente, que es mejor que corromper datos en silencio).

Puedes lanzar excepciones del sistema (`ArgumentException`, `InvalidOperationException`) o crear las tuyas. Para crear un tipo propio, heredas de `Exception`:

```

class FondosInsuficientesException : Exception
{
    public decimal MontoSolicitado { get; }
    public decimal SaldoDisponible { get; }

    public FondosInsuficientesException(string mensaje, decimal solicitado, decimal
disponible)
        : base(mensaje)
    {
        MontoSolicitado = solicitado;
        SaldoDisponible = disponible;
    }
}

// Uso:
if (monto > Saldo)
{
    throw new FondosInsuficientesException(
        "No tienes saldo suficiente.",
        monto,
        Saldo);
}

// Y quien la atrapa puede usar los datos extra:
catch (FondosInsuficientesException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine("Pediste: $" + ex.MontoSolicitado + ", tienes: $" +
ex.SaldoDisponible);
}

```

¿Cuándo vale la pena crear una excepción propia? Cuando el error representa una condición ESPECÍFICA DE TU NEGOCIO que quieres distinguir de los errores genéricos. Un banco distingue "fondos insuficientes" de "cuenta bloqueada" de "error de red": son situaciones distintas que requieren respuestas distintas. Si todo fuera `Exception` genérica, no podrías reaccionar diferente a cada caso.

También puedes RE-LANZAR una excepción desde un catch cuando la atrapas para hacer algo (como registrarla) pero no puedes resolverla:

```
try
{
    GuardarVenta(venta);
}
catch (IOException ex)
{
    RegistrarEnLog("Falló el guardado: " + ex.Message);
    throw; // re-lanza la MISMA excepción para que suba a quien sí pueda manejarla
}
```

Ojo: se escribe `throw;` solo, sin la variable. Así se conserva el `StackTrace` original. Si escribes `throw ex;`, el `StackTrace` se reinicia y pierdes el rastro del origen real. Detalle fino, pero profesional.

Cuándo atrapar y cuándo NO atrapar: el criterio

Aquí está la lección más importante de la clase, la que separa al principiante del profesional:

ATRAPA una excepción cuando:

- Puedes HACER ALGO ÚTIL al respecto: pedir el dato de nuevo, usar un valor por defecto sensato, reintentar la operación, guardar un respaldo, mostrar un mensaje que el usuario entienda y ofrecerle una salida.
- Estás en la FRONTERA del programa: la interfaz de usuario, el punto de entrada de una petición, el ciclo principal. Ahí sí pon una red de seguridad para que nada truene feo frente al usuario.

NO ATRAPES (deja que suba) cuando:

- No tienes ni idea de qué hacer con ella. Atrapar para no hacer nada (catch vacío) es el peor pecado: esconde el error y el programa sigue con datos corruptos. Es como desconectar la alarma de incendios porque te molesta el ruido.
- Es un error de PROGRAMACIÓN (bug): `NullReferenceException` por no verificar null, `IndexOutOfRangeException` por mal cálculo. Esos se ARREGLAN en el código, no se atrapan. Atraparlos es ponerle curita a una fractura.
- Solo quieres "que no truene". El programa que nunca truena pero da resultados incorrectos es PEOR que el que truena: el que truena al menos avisa.

El error clásico del principiante es este:

```
// MAL: la trampa del "por si acaso"
try
{
    // ... 200 líneas de código ...
}
catch (Exception)
{
    // no hace nada. El error desaparece como por arte de magia.
}
```

Esto es una aspiradora de errores: se traga todo, no deja rastro, y cuando el programa se comporta raro nadie sabe por qué. Si vas a atrapar `Exception` genérica, que sea en la frontera del programa, REGISTRANDO el error:

```
// BIEN: red de seguridad en la frontera, con registro
try
{
    EjecutarPrograma();
}
catch (Exception ex)
{
    File.AppendAllText("errores.log",
        DateTime.Now + " - " + ex.GetType().Name + ": " + ex.Message + "\n" +
        ex.StackTrace + "\n");
    Console.WriteLine("Ocurrió un error inesperado. Ya quedó registrado para revisarlo.");
}
```

Regla de oro: atrapa lo ESPECÍFICO donde PUEDES REACCIONAR, y deja una red genérica CON REGISTRO solo en la frontera. Y nunca, nunca, un catch vacío.

TryParse: la alternativa elegante para conversiones

Un caso especial merece mención: convertir texto a número. Ya viste que `int.Parse` lanza `FormatException` si el texto no es número. Para entrada del usuario, la forma profesional NO es try/catch: es `TryParse`, que devuelve true/false en vez de tronar:

```

Console.Write("Edad: ");
string texto = Console.ReadLine();

int edad;
if (int.TryParse(texto, out edad))
{
    Console.WriteLine("Tienes " + edad + " años.");
}
else
{
    Console.WriteLine("'" + texto + "' no es una edad válida.");
}

```

`TryParse` existe para int, double, decimal, DateTime, bool y más. Úsalo SIEMPRE que conviertas datos que vienen del usuario o de fuera de tu programa. Reserva `Parse` para cuando estás seguro del formato (por ejemplo, convirtiendo datos que tú mismo guardaste). Es el mismo patrón que `TryGetValue` en diccionarios: intentar sin tronar.

Filtros de excepción con when: atrapar con condición

A veces quieres atrapar una excepción solo SI se cumple una condición extra. Por ejemplo: atrapar `IOException` solo cuando el disco está lleno, pero dejar pasar otros errores de disco. C# tiene los filtros de excepción con la palabra `when`:

```

try
{
    DescargarArchivo(url);
}
catch (IOException ex) when (ex.Message.Contains("espacio"))
{
    Console.WriteLine("El disco está lleno. Libera espacio e intenta de nuevo.");
}
catch (IOException ex)
{
    Console.WriteLine("Error de disco: " + ex.Message);
}

```

El primer catch solo atiende la excepción si ADEMÁS se cumple la condición del `when`; si no se cumple, C# lo ignora como si no existiera y pasa al siguiente catch. Es más elegante que atrapar y luego hacer `if` adentro para re-lanzar.

Otro uso típico: reintentar una operación un número limitado de veces. Imagina que guardar en red falla de forma intermitente:

```
int intentos = 0;
bool guardado = false;

while (!guardado && intentos < 3)
{
    try
    {
        GuardarEnRed(datos);
        guardado = true;
    }
    catch (IOException ex) when (intentos < 2)
    {
        intentos++;
        Console.WriteLine($"Fallo el intento {intentos}, reintentando...
({ex.Message})");
        System.Threading.Thread.Sleep(1000); // espera 1 segundo antes de
reintentar
    }
}

if (!guardado)
{
    Console.WriteLine("No se pudo guardar después de 3 intentos.");
}
```

El filtro `when (intentos < 2)` hace que los dos primeros fallos se atrapen y reintenten, pero el tercero (cuando intentos ya vale 2) NO se atrapa aquí: la excepción sube a quien pueda manejarla de verdad. Es un patrón de "reintento con límite" que verás en sistemas que hablan con redes, bases de datos o APIs: fallar una vez puede ser mala suerte, fallar tres veces seguidas es un problema real.

Nota que los filtros `when` no son lo primero que debes aprender de memoria; son una herramienta de precisión para cuando los catch simples no bastan. Pero cuando los necesites, nada más los sustituye con tanta elegancia.

Clase 23: Archivos, datos que sobreviven al apagar la computadora

Por qué los archivos importan: la memoria es volátil

Todo lo que guardas en variables vive en la memoria RAM, y la RAM se borra cuando el programa termina. Es como escribir en un pizarrón: sirve mientras estás en el salón, pero al salir alguien lo borra. Los archivos en disco son el cuaderno: lo que escribes ahí sigue existiendo mañana, la próxima semana y después de reiniciar la computadora.

Los usos son infinitos: guardar la configuración del programa, llevar un registro (log) de errores, exportar reportes, guardar las notas del usuario, leer datos de entrada por lotes. Todo programa serio lee o escribe archivos. Y C# lo hace muy fácil con el espacio de nombres ``System.IO``.

File: lectura y escritura en dos líneas

La clase estática ``File`` tiene los métodos más directos. Para archivos de texto pequeños o medianos (configuraciones, notas, logs), son todo lo que necesitas:

```

using System.IO;

// Escribir: crea el archivo si no existe, lo SOBREScribe si existe
File.WriteAllText("nota.txt", "Hola, este es el contenido de la nota.");

// Leer: devuelve TODO el contenido como un string
string contenido = File.ReadAllText("nota.txt");
Console.WriteLine(contenido);

// Agregar al final sin borrar lo anterior
File.AppendAllText("nota.txt", "\nSegunda línea agregada después.");

// Preguntar si existe ANTES de leer (evita FileNotFoundException)
if (File.Exists("nota.txt"))
{
    string texto = File.ReadAllText("nota.txt");
}
else
{
    Console.WriteLine("El archivo no existe todavía.");
}

// También hay versiones para arreglos de líneas
string[] lineas = { "Línea 1", "Línea 2", "Línea 3" };
File.WriteAllLines("lineas.txt", lineas);           // escribe cada elemento en una
línea
string[] leidas = File.ReadAllLines("lineas.txt"); // lee y devuelve un arreglo de
líneas

// Copiar, mover y borrar
File.Copy("nota.txt", "respaldo.txt");           // copia
File.Move("nota.txt", "nota_final.txt");         // mueve/renombra
File.Delete("respaldo.txt");                      // borra (¡sin papelera, cuidado!)

```

Tres advertencias de colega:

1. `WriteAllText` SOBREScribe sin preguntar. Si el archivo tenía información valiosa y lo sobreescribes, se perdió. Para agregar sin borrar usa `AppendAllText`. Para no pisar nada sin querer, verifica `Exists` primero o guarda respaldos.
2. `File.Delete` borra de verdad: no va a la papelera de reciclaje. En tus programas, pide confirmación al usuario antes de borrar, o mejor: mueve a una carpeta de "papelera" en vez de borrar.

3. Estos métodos trabajan con el archivo COMPLETO en memoria. Para un archivo de 500 MB, `ReadAllText`` intentará cargarlo todo a la RAM. Para archivos grandes usa `StreamReader`, que lee por partes (viene en la siguiente sección).

Rutas: que tu programa funcione en cualquier computadora

El error clásico del principiante: escribir rutas absolutas de SU computadora en el código:

```
// MAL: solo funciona en tu máquina
string contenido =
File.ReadAllText("C:\\Usuarios\\TuNombre\\Documentos\\datos.txt");
```

En la computadora del usuario esa carpeta no existe y el programa truena. Las soluciones profesionales:

1. Rutas RELATIVAS: si el archivo está junto al programa (o en una subcarpeta), usa rutas relativas. `"datos.txt"` busca en la carpeta donde corre el programa; `"datos\\notas.txt"` en la subcarpeta datos. Así funciona en cualquier máquina mientras la estructura de carpetas viaje con el programa.
2. `Path.Combine``: arma rutas uniendo partes con el separador correcto del sistema. En Windows el separador es `\\`, en Linux/Mac es `/`. Si concatenas a mano con `\\`, tu programa falla en otros sistemas. `Path.Combine`` lo resuelve solo:

```
string carpeta = "datos";
string archivo = "notas.txt";
string ruta = Path.Combine(carpeta, archivo); // "datos\\notas.txt" en Windows,
"datos/notas.txt" en Linux
```

3. Carpetas especiales del sistema: para datos del usuario, no inventes rutas; usa las que el sistema te da:

```
string escritorio = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
string documentos =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
string datosApp =
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);

string rutaNotas = Path.Combine(documentos, "MisNotas", "nota.txt");
```

4. `Directory`` para manejar carpetas: crear la carpeta antes de guardar (si no existe, `WriteAllText`` truena porque la carpeta no está):

```

string carpeta = Path.Combine(documentos, "MisNotas");
Directory.CreateDirectory(carpeta); // la crea si no existe; si existe no hace nada
(no trueno)
string ruta = Path.Combine(carpeta, "nota.txt");
File.WriteAllText(ruta, "Contenido");

// Otros útiles:
bool existe = Directory.Exists(carpeta);
string[] archivos = Directory.GetFiles(carpeta); // lista de archivos en la carpeta
string[] archivosTxt = Directory.GetFiles(carpeta, "*.txt"); // solo los .txt

```

Regla práctica: SIEMPRE que vayas a escribir en una subcarpeta, llama `Directory.CreateDirectory` primero. Es idempotente (no pasa nada si ya existe) y te ahorra la excepción más tonta de esta clase.

StreamReader y StreamWriter: control fino y archivos grandes

`File.ReadAllText` es cómodo pero todo-o-nada. Cuando necesitas leer línea por línea (archivos grandes, logs, procesar mientras lees) o escribir por partes, usas streams: flujos de datos.

```

using System.IO;

// ESCRIBIR por partes con StreamWriter
using (StreamWriter escritor = new StreamWriter("bitacora.txt", append: true))
{
    escritor.WriteLine(DateTime.Now + " - El programa inició.");
    escritor.WriteLine(DateTime.Now + " - Usuario conectado.");
    escritor.WriteLine(DateTime.Now + " - Proceso terminado.");
}
// Al salir del using, el archivo se cierra SOLO, aunque haya tronado algo adentro

// LEER línea por línea con StreamReader (ideal para archivos grandes)
using (StreamReader lector = new StreamReader("bitacora.txt"))
{
    string linea;
    while ((linea = lector.ReadLine()) != null)
    {
        Console.WriteLine("Leído: " + linea);
        // Aquí puedes procesar cada línea sin cargar todo el archivo a memoria
    }
}

```

El patrón ``while ((línea = lector.ReadLine()) != null)`` es un clásico que verás en todas partes: ``ReadLine`` devuelve null cuando se acaba el archivo, así que el ciclo lee hasta el final. Memoriza su forma: asigna y compara con null en la misma condición.

using: el guardián de los recursos

Fíjate en la palabra ``using`` con paréntesis (distinta del ``using System.IO;`` de arriba, que es para espacios de nombres). Este ``using`` dice: "crea este objeto, úsalo dentro de estas llaves, y al salir CIÉRRALO Y LIBÉRALO, pase lo que pase".

¿Por qué importa? Porque los archivos abiertos son recursos del sistema operativo: hay un límite de cuántos puedes tener abiertos, y un archivo abierto por tu programa puede quedar BLOQUEADO para otros programas. Si tu programa truena a la mitad de escribir y no cierra el archivo, queda bloqueado hasta que reinicies o el sistema lo libere. Es como salir de tu casa dejando la llave puesta por dentro: nadie más puede entrar.

Sin ``using`` tendrías que escribir el `try/finally` a mano:

```
// Lo que using hace por ti automáticamente (no lo escribas así, es solo para
entender):
StreamWriter escritor = null;
try
{
    escritor = new StreamWriter("bitacora.txt", append: true);
    escritor.WriteLine("Hola");
}
finally
{
    if (escritor != null)
    {
        escritor.Close(); // o escritor.Dispose();
    }
}
```

El ``using`` es exactamente eso, pero en una línea. Úsalo SIEMPRE con: archivos (`StreamReader/StreamWriter`), conexiones a bases de datos, y en general cualquier objeto que implemente ``IDisposable`` (¿te suena? es la interfaz de la unidad 7: el contrato de "tengo recursos que liberar"). De hecho, ``using`` solo funciona con objetos que implementan ``IDisposable``: el compilador lo exige.

Errores comunes con archivos:

1. No usar ``using`` y dejar archivos bloqueados. El síntoma: "el programa funcionó una vez y después dice que el archivo está en uso". La cura: ``using``, siempre.
2. Rutas absolutas hardcodeadas. Funciona en tu máquina, truena en todas las demás. Usa rutas relativas, ``Path.Combine`` y ``Environment.SpecialFolder``.
3. No crear la carpeta antes de escribir. ``File.WriteAllText`` crea el ARCHIVO pero no las CARPETAS. ``Directory.CreateDirectory`` primero.
4. Leer sin verificar ``Exists``. O atrapar la ``FileNotFoundException``, o verificar antes. Para archivos de configuración que "deberían" existir, verificar y crear uno por defecto es más amable que tronar.
5. Suponer la codificación del texto. ``ReadAllText`/`WriteAllText`` usan UTF-8 por defecto, que maneja acentos y ñ sin problema. Si lees un archivo creado por un programa viejo en otra codificación y ves símbolos raros (la "mojibake"), el problema es la codificación, no tu código. Las versiones de estos métodos aceptan un parámetro ``Encoding`` para esos casos.

Clase 24: Cadenas avanzadas, el pan de cada día del programador

Las cadenas son inmutables (y por qué te debe importar)

Primera verdad fundamental: en C#, los ``string`` son INMUTABLES. Una vez creada una cadena, no se puede modificar. Cada vez que "modificas" una cadena, en realidad estás creando una NUEVA y tirando la vieja:

```
string saludo = "Hola";
saludo += " mundo"; // NO modifica "Hola": crea "Hola mundo" nuevo y lo asigna
saludo = saludo.ToUpper(); // crea OTRO nuevo: "HOLA MUNDO"
```

Para una o diez operaciones no pasa nada. Para un ciclo que concatena 100,000 veces, estás creando 100,000 cadenas temporales y el recolector de basura trabaja horas extra: el programa se pone lento y come memoria. Ese es el famoso problema de "concatenar en ciclos", y su solución se llama ``StringBuilder`` (llegamos ahí en un momento). Por ahora quédate con la idea: las cadenas no se modifican, se reemplazan.

Interpolación: la forma moderna de armar mensajes

Antes se armaban mensajes concatenando con ``+``, que es tedioso y propenso a errores:

```
// Forma vieja (funciona, pero fea):
string mensaje = "Hola " + nombre + ", tienes " + edad + " años y tu saldo es $" +
saldo + ".";

// Forma moderna: interpolación con $
string mensaje2 = $"Hola {nombre}, tienes {edad} años y tu saldo es ${saldo}.";
```

Con `\$` antes de las comillas, lo que está entre `{}` se evalúa y se inserta. Se lee como la frase final, no como un rompecabezas. Además puedes poner expresiones adentro:

```
int a = 5, b = 3;
Console.WriteLine($"La suma de {a} + {b} es {a + b}."); // La suma de 5 + 3 es 8.
Console.WriteLine($"Mañana será {(DateTime.Now.Day + 1)} del mes.");
```

Y lo mejor: puedes combinar interpolación con FORMATO (dos puntos dentro de las llaves):

```
decimal precio = 1234.5m;
Console.WriteLine($"Precio: {precio:C}");           // Precio: $1,234.50 (moneda
local)
Console.WriteLine($"Precio: {precio:N2}");          // Precio: 1,234.50 (número con 2
decimales)
Console.WriteLine($"Descuento: {0.15:P0}");        // Descuento: 15 % (porcentaje sin
decimales)

DateTime hoy = DateTime.Now;
Console.WriteLine($"Hoy es {hoy:d}");               // fecha corta: 19/09/2026
Console.WriteLine($"Hoy es {hoy:D}");               // fecha larga: sábado, 19 de
septiembre de 2026
Console.WriteLine($"Hora: {hoy:HH:mm}");            // Hora: 14:35 (formato
personalizado)
```

Los especificadores más usados para números: `C` (moneda), `N` (número con separadores de miles), `P` (porcentaje), `F` (punto fijo), `D` para enteros (dígitos mínimos, útil para folios: `{7:D5}` -> "00007"). Para fechas: `d` (corta), `D` (larga), `t`/`T` (hora), o formatos personalizados como `yyyy-MM-dd` (el formato universal para guardar fechas en archivos: "2026-09-19", que además se ordena alfabéticamente en orden cronológico).

Ojo con un detalle cultural: el formato de moneda y fechas depende de la configuración regional de la computadora. `{precio:C}` muestra \$ en México y algo distinto en España. Para archivos o datos que viajan entre sistemas, usa formatos explícitos e invariantes en vez de depender de la cultura local.

StringBuilder: para armar textos grandes

Cuando vas a concatenar muchas veces (en un ciclo, armando un reporte, generando un archivo), usa `StringBuilder` (espacio de nombres `System.Text`). Es un "constructor de cadenas" mutable: tiene un búfer interno que crece, y solo al final produces la cadena:

```
using System.Text;

StringBuilder reporte = new StringBuilder();
reporte.AppendLine("=== REPORTE DE VENTAS ===");
reporte.AppendLine("Fecha: " + DateTime.Now.ToString("yyyy-MM-dd"));

foreach (var v in ventas)
{
    // AppendLine agrega el texto más un salto de línea
    reporte.AppendLine($"{v.Producto}: {v.Cantidad} x ${v.PrecioUnitario} = ${v.Ingreso}");
}

reporte.AppendLine($"TOTAL: ${ventas.Sum(v => v.Ingreso)}");

string textoFinal = reporte.ToString(); // hasta aquí se crea la cadena definitiva
File.WriteAllText("reporte.txt", textoFinal);
```

Métodos clave: `Append` (agrega sin salto), `AppendLine` (agrega con salto), `AppendFormat` (como `string.Format`), `Clear` (vacía el búfer para reusarlo), `ToString` (produce el resultado final). También puedes darle capacidad inicial: `new StringBuilder(10000)` si sabes que el texto será grande.

¿Cuándo usar cada uno? Regla simple: si son unas cuantas concatenaciones (menos de ~10) o una interpolación, usa `string` normal con `$`. Si es un ciclo con muchas iteraciones o estás armando un documento, usa `StringBuilder`. El principiante peca por no usarlo; el obsesivo peca por usarlo hasta para juntar dos palabras. El punto medio profesional: ciclos y textos grandes -> `StringBuilder`; lo demás -> interpolación.

El error clásico que esto evita:


```
// MAL: 10,000 cadenas temporales, lento y glotón de memoria
string resultado = "";
for (int i = 0; i < 10000; i++)
{
    resultado += "línea " + i + "\n";
}

// BIEN: un solo búfer, rápido
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.AppendLine("línea " + i);
}
string resultado2 = sb.ToString();
```

La caja de herramientas del texto: Split, Join, Contains y compañía

Estos métodos los usarás TODOS LOS DÍAS. Vamos uno por uno con ejemplos prácticos:

Split: partir una cadena en pedazos. Perfecto para procesar CSV, listas separadas por comas, rutas:

```
string csv = "Ana,25,CDMX,Ingeniera";
string[] partes = csv.Split(',');
// partes = ["Ana", "25", "CDMX", "Ingeniera"]
string nombre = partes[0]; // "Ana"

// Con varias opciones: quitar entradas vacías (por si hay comas dobles)
string datos = "a,,b,c";
string[] limpio = datos.Split(',', StringSplitOptions.RemoveEmptyEntries); // ["a", "b", "c"]
```

Join: lo contrario, unir pedazos en una cadena. Ya lo viste en la unidad 7 con `string.Join(", ", lista)`:

```
string[] frutas = { "manzana", "pera", "uva" };
string lista = string.Join(", ", frutas); // "manzana, pera, uva"
string csv2 = string.Join(";", frutas); // "manzana;pera;uva"
```

Nota: `Join` es ESTÁTICO (se llama sobre la clase `string`, no sobre una instancia). Es el que más se olvida ese detalle.

Contains, StartsWith, EndsWith: preguntas sobre el contenido:

```
string titulo = "Curso Completo de C#";

bool hablaDeCSharp = titulo.Contains("C#");           // true
bool empiezaConCurso = titulo.StartsWith("Curso");    // true
bool terminaConJava = titulo.EndsWith("Java");        // false

// Búsqueda insensible a mayúsculas (truco profesional):
bool contiene = titulo.Contains("curso", StringComparison.OrdinalIgnoreCase); // true
```

Ese segundo parámetro `StringComparison.OrdinalIgnoreCase` evita el truco del `ToLower()` y es más correcto: compara sin importar mayúsculas de forma explícita y eficiente. Úsalo en búsquedas de texto que el usuario escribe.

Replace y Substring: transformar pedazos:

```
string texto = "Hola mundo cruel";
string amable = texto.Replace("cruel", "maravilloso"); // "Hola mundo maravilloso"

string primeros4 = texto.Substring(0, 4); // "Hola" (desde 0, toma 4)
string desde5 = texto.Substring(5);       // "mundo cruel" (desde 5 hasta el final)
```

Cuidado con `Substring`: si el índice o la longitud se salen del rango, lanza `ArgumentOutOfRangeException`. Verifica longitudes antes, sobre todo con texto que viene del usuario.

Trim, ToUpper, ToLower: limpieza básica:

```
string sucio = "  hola mundo  ";
string limpio2 = sucio.Trim();           // "hola mundo" (quita espacios de los extremos)
string sinIzq = sucio.TrimStart();       // quita solo a la izquierda
string sinDer = sucio.TrimEnd();         // quita solo a la derecha

string grito = limpio2.ToUpper();        // "HOLA MUNDO"
string susurro = limpio2.ToLower();      // "hola mundo"
```

`Trim()` es OBLIGATORIO con entrada del usuario: la gente deja espacios sin querer (" Ana "), y " Ana " != "Ana" para el programa. Limpia siempre lo que lees de consola o de archivos antes de comparar.

IndexOf y LastIndexOf: encontrar POSICIONES:

```
string email = "usuario@ejemplo.com";
int arroba = email.IndexOf('@'); // 7
bool esEmail = arroba > 0 && email.IndexOf('.', arroba) > arroba; // validación
ingenua

string dominio = email.Substring(arroba + 1); // "ejemplo.com"
```

Si no encuentra el carácter, devuelve -1. Ese -1 es tu señal de "no existe": verifícalo antes de usar la posición en un Substring.

Length y el indexador: la cadena también se puede recorrer como arreglo de caracteres:

```
string palabra = "hola";
Console.WriteLine(palabra.Length); // 4
Console.WriteLine(palabra[0]);      // 'h'
foreach (char c in palabra)
{
    Console.WriteLine(c);
}
```

Validar texto: IsNullOrEmpty e IsNullOrWhiteSpace

Antes de procesar una cadena que viene de fuera, válidala. `string` trae dos guardianes estáticos:

```
string entrada = Console.ReadLine();

if (string.IsNullOrEmpty(entrada))
{
    Console.WriteLine("No escribiste nada. Intenta de nuevo.");
    return;
}
// IsNullOrWhiteSpace: true si es null, vacía ("") o solo espacios (" ")
// IsNullOrEmpty: true si es null o vacía (pero " " pasa como válida)

string nombreLimpio = entrada.Trim(); // ahora sí, a trabajar
```

Usa `IsNullOrWhiteSpace` casi siempre: cubre el caso del usuario que solo apretó la barra espaciadora. Es la primera línea de defensa de cualquier programa que lee texto.

Parse vs TryParse (repaso aplicado a texto)

Ya viste `TryParse` en la clase 22; aquí va el contexto completo de conversiones texto -> dato:

```
string textoEdad = "25";
int edad = int.Parse(textoEdad);           // 25. Truena si no es número.

string textoPrecio = "199.99";
decimal precio;
if (decimal.TryParse(textoPrecio, out precio)) // true/false, sin tronar
{
    Console.WriteLine($"Precio válido: {precio:C}");
}

string textoFecha = "2026-09-19";
DateTime fecha;
if (DateTime.TryParse(textoFecha, out fecha))
{
    Console.WriteLine($"Fecha: {fecha:D}");
}

// Convertir dato -> texto con formato:
string folio = 42.ToString("D6");           // "000042"
string porcentaje = 0.857.ToString("P1");   // "85.7 %"
```

Para fechas que TÚ guardas en archivos, usa el formato universal `yyyy-MM-dd` al guardar y `TryParse` al leer: evita ambigüedades entre "03/04/2026" (¿3 de abril o 4 de marzo? depende del país).

Comparar cadenas como profesional: StringComparison

Comparar texto parece trivial hasta que te topas con los casos reales: ¿"Ana" es igual a "ANA"? ¿"café" es igual a "CAFE"? ¿En qué orden van "n10" y "n9"? C# te da control total con el enumerador `StringComparison`:

```

string a = "Ana";
string b = "ANA";

// Igualdad insensible a mayúsculas (la correcta, sin ToLower)
bool iguales = a.Equals(b, StringComparison.OrdinalIgnoreCase); // true

// Ordenamiento para mostrar al usuario (respeta el idioma: la ñ va donde debe)
string[] nombres = { "Óscar", "ana", "Ñoño", "beto" };
Array.Sort(nombres, StringComparer.CurrentCultureIgnoreCase);
// ordenado según las reglas del español

// Ordenamiento para claves internas, rápido y predecible (byte por byte)
Array.Sort(nombres, StringComparer.Ordinal);

```

Las opciones principales:

- ``Ordinal``: compara byte por byte, rapidísimo y 100 por ciento predecible. Úsalo para claves internas, identificadores, nombres de archivo que el programa maneja solo, tokens de seguridad. Es la opción por defecto de ``==`` y de ``Dictionary<string, ...>``.
- ``OrdinalIgnoreCase``: igual pero sin importar mayúsculas. Tu opción para comparar lo que escribe el usuario con valores internos ("s" vs "S", comandos, respuestas sí/no).
- ``CurrentCulture`` / ``CurrentCultureIgnoreCase``: compara con las reglas del idioma del usuario. Úsalo para texto que el usuario VA A VER ordenado o comparado: listas de nombres, búsquedas en catálogos. En español, sabe que "ch" y "ñ" tienen su lugar.
- ``InvariantCulture``: como `CurrentCulture` pero con reglas fijas independientes de la máquina. Para datos que se guardan o viajan entre sistemas.

¿Por qué no usar siempre ``ToLower()`` para comparar sin mayúsculas? Tres razones: crea cadenas temporales (basura para el recolector), es más lento, y en algunos idiomas (el turco es el ejemplo clásico: su "i" mayúscula no es "I") da resultados incorrectos. ``Equals`` con ``StringComparison`` no tiene ninguno de esos problemas.

Y un caso especial: los diccionarios con claves insensibles a mayúsculas. Si tu agenda debe tratar "Ana" y "ANA" como la misma persona sin andar normalizando a mano, el diccionario acepta el comparador en el constructor:

```

var agenda = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);
agenda.Add("Ana", "555-0101");
Console.WriteLine(agenda["ANA"]); // 555-0101, ¡funciona!
bool existe = agenda.ContainsKey("ana"); // true

```

Así el diccionario mismo se encarga de la insensibilidad: más limpio que el ``ToLower()`` manual que usamos en el ejemplo de la agenda, y sin riesgo de olvidar normalizar en algún método. Cuando diseñes con diccionarios de texto, decide desde el inicio si las claves distinguen mayúsculas; cambiarlo después es un dolor de cabeza.

Errores comunes con cadenas

1. Concatenar en ciclos con ``+=``: ya lo viste, usa ``StringBuilder``. El síntoma es un programa que "se pone lento con muchos datos".
2. No limpiar la entrada del usuario: comparar ``"Ana"`` con ``"Ana "`` falla. ``Trim()`` siempre después de ``ReadLine()``.
3. Comparar cadenas con ``==`` funciona en C# (compara contenido, no referencias), pero para búsquedas insensibles a mayúsculas usa ``Equals`` con ``StringComparison`` o ``Contains`` con ``StringComparison.OrdinalIgnoreCase``, no el truco del ``ToLower()`` en ambos lados (que crea cadenas temporales y puede fallar con ciertas culturas).
4. ``Substring`` sin verificar: con texto más corto de lo esperado, lanza excepción. Verifica ``Length`` primero o usa ``Math.Min``.
5. Suponer que ``Split`` siempre devuelve las partes que esperas: si el usuario escribe "Ana,25" sin la ciudad, ``partes[2]`` truena con ``IndexOutOfRangeException``. Verifica ``partes.Length`` antes de indexar. Los datos de afuera siempre vienen "chuecos": programa a la defensiva.
6. Olvidar que los métodos de string DEVUELVEN cadenas nuevas: ``texto.ToUpper();`` solo NO cambia ``texto``. Tienes que asignar: ``texto = texto.ToUpper();``. Por la inmutabilidad, si no asignas el resultado, el trabajo se pierde en el aire. Este bug es silencioso y muy común.

Ejemplos paso a paso

Ejemplo 1: Bloc de notas que guarda y lee en disco

Un programa de consola para escribir notas, guardarlas en archivos, listarlas y leerlas. Junta archivos, excepciones, cadenas y menús. Es el ejemplo más completo de la unidad: léelo con calma.

```
using System;
using System.IO;
using System.Linq;
```

```

class BlocDeNotas
{
    private string carpeta;

    public BlocDeNotas()
    {
        // Las notas viven en Documentos/MisNotas, en CUALQUIER computadora
        string documentos =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        carpeta = Path.Combine(documentos, "MisNotas");
        Directory.CreateDirectory(carpeta); // la crea si no existe
    }

    public void CrearNota()
    {
        Console.WriteLine("Título de la nota (sin espacios raros): ");
        string titulo = Console.ReadLine().Trim();

        if (string.IsNullOrEmpty(titulo))
        {
            Console.WriteLine("El título no puede estar vacío.");
            return;
        }

        Console.WriteLine("Escribe tu nota (línea vacía para terminar:");

        // StringBuilder para armar el texto línea por línea sin concatenar en ciclo
        System.Text.StringBuilder contenido = new System.Text.StringBuilder();
        string linea;
        while (!string.IsNullOrEmpty(linea = Console.ReadLine()))
        {
            contenido.AppendLine(linea);
        }

        string ruta = Path.Combine(carpeta, titulo + ".txt");

        try
        {
            if (File.Exists(ruta))
            {
                Console.WriteLine("Ya existe una nota con ese título. ¿Sobrescribir?
(s/n): ");
                if (Console.ReadLine().Trim().ToLower() != "s")
                {

```

```

        Console.WriteLine("Nota no guardada.");
        return;
    }
}

File.WriteAllText(ruta, contenido.ToString());
Console.WriteLine("Nota guardada en: " + ruta);
}
catch (IOException ex)
{
    Console.WriteLine("No se pudo guardar la nota: " + ex.Message);
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Sin permiso para escribir ahí: " + ex.Message);
}
}

public void ListarNotas()
{
    string[] archivos = Directory.GetFiles(carpeta, "*.txt");

    if (!archivos.Any())
    {
        Console.WriteLine("No hay notas guardadas.");
        return;
    }

    Console.WriteLine("--- Tus notas (" + archivos.Length + ") ---");
    foreach (string ruta in archivos.OrderBy(r => r))
    {
        // Path.GetFileNameWithoutExtension: solo el nombre, sin carpeta ni .txt
        string titulo = Path.GetFileNameWithoutExtension(ruta);
        DateTime modificado = File.GetLastWriteTime(ruta);
        Console.WriteLine("- " + titulo + " (modificada: " +
modificado.ToString("g") + ")");
    }
}

public void LeerNota()
{
    Console.Write("Título de la nota a leer: ");
    string titulo = Console.ReadLine().Trim();
    string ruta = Path.Combine(carpeta, titulo + ".txt");

```



```

try
{
    string contenido = File.ReadAllText(ruta);
    Console.WriteLine("=== " + titulo + " ===");
    Console.WriteLine(contenido);

    // Dato curioso con cadenas: cuántas palabras tiene
    int palabras = contenido.Split(
        new char[] { ' ', '\n', '\r', '\t' },
        StringSplitOptions.RemoveEmptyEntries).Length;
    Console.WriteLine("--- (" + palabras + " palabras) ---");
}
catch (FileNotFoundException)
{
    Console.WriteLine("No existe una nota llamada '" + titulo + "'.");
}
catch (IOException ex)
{
    Console.WriteLine("No se pudo leer la nota: " + ex.Message);
}
}

public void EliminarNota()
{
    Console.Write("Título de la nota a eliminar: ");
    string titulo = Console.ReadLine().Trim();
    string ruta = Path.Combine(carpeta, titulo + ".txt");

    if (!File.Exists(ruta))
    {
        Console.WriteLine("No existe una nota llamada '" + titulo + "'.");
        return;
    }

    Console.Write("¿Seguro que quieres eliminar '" + titulo + "'? (s/n): ");
    if (Console.ReadLine().Trim().ToLower() == "s")
    {
        try
        {
            File.Delete(ruta);
            Console.WriteLine("Nota eliminada.");
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo eliminar: " + ex.Message);
        }
    }
}

```

```

        }
    }
    else
    {
        Console.WriteLine("Eliminación cancelada.");
    }
}
}

class Program
{
    static void Main()
    {
        BlocDeNotas bloc = new BlocDeNotas();
        string opcion = "";

        // Red de seguridad en la frontera: si algo inesperado truena, no se cierra
        try
        {
            while (opcion != "5")
            {
                Console.WriteLine();
                Console.WriteLine("=== BLOC DE NOTAS ===");
                Console.WriteLine("1. Crear nota");
                Console.WriteLine("2. Listar notas");
                Console.WriteLine("3. Leer nota");
                Console.WriteLine("4. Eliminar nota");
                Console.WriteLine("5. Salir");
                Console.Write("Opción: ");
                opcion = Console.ReadLine().Trim();

                if (opcion == "1") bloc.CrearNota();
                else if (opcion == "2") bloc.ListarNotas();
                else if (opcion == "3") bloc.LeerNota();
                else if (opcion == "4") bloc.EliminarNota();
                else if (opcion != "5") Console.WriteLine("Opción no válida.");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Ocurrió un error inesperado: " + ex.Message);
            Console.WriteLine("Tus notas están a salvo en tu carpeta de documentos.");
        }
    }
}

```

```

        Console.WriteLine(";Hasta luego!");
    }
}

```

Explicación de las decisiones clave:

- El constructor arma la ruta con ``Environment.SpecialFolder.MyDocuments`` + ``Path.Combine`` + ``Directory.CreateDirectory``: funciona en cualquier computadora y nunca truena por carpeta inexistente. Esto es programar portable desde el diseño.
- ``CrearNota`` valida el título con ``IsNullOrWhiteSpace`` (no deja títulos vacíos), arma el contenido con ``StringBuilder`` (buena práctica en vez de concatenar en el ciclo), y ANTES de sobrescribir pregunta al usuario: respeta sus datos.
- Los `catch` son ESPECÍFICOS: ``IOException`` para problemas de disco, ``UnauthorizedAccessException`` para permisos. Cada uno con mensaje útil. No hay un ``catch (Exception)`` tragatodo en la lógica; la red genérica está solo en el Main, en la frontera.
- ``ListarNotas`` usa ``Directory.GetFiles(carpeta, "*.txt")`` para obtener solo notas, ``Path.GetFileNameWithoutExtension`` para mostrar títulos limpios, y ``File.GetLastWriteTime`` para la fecha de modificación con formato ``"g"`` (fecha+hora corta).
- ``LeerNota`` atrapa ``FileNotFoundException`` por separado para dar el mensaje amable "no existe" en vez del técnico. Y de paso cuenta palabras con ``Split`` usando varios separadores (espacio, saltos de línea, tabulación) y ``RemoveEmptyEntries``.
- ``EliminarNota`` pide confirmación explícita antes de ``File.Delete`` (que no tiene papelera). Programar a la defensiva también es cuidar los datos del usuario.
- El ``try`` del Main envuelve el ciclo completo como red de seguridad final: si algo rarísimo truena, el usuario ve un mensaje tranquilo y la certeza de que sus notas siguen en disco.

Ejemplo 2: Procesador de texto (analizar, buscar y reemplazar)

Un programa que lee un archivo de texto y lo analiza: cuenta palabras, líneas y caracteres, muestra las palabras más frecuentes, busca frases y reemplaza texto guardando el resultado. Puro manejo de cadenas con propósito real.

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

```

```

class ProcesadorTexto
{
    private string contenido = "";
    private string rutaArchivo = "";

    public bool CargarArchivo()
    {
        Console.Write("Ruta del archivo de texto: ");
        rutaArchivo = Console.ReadLine().Trim().Trim('"'); // quita comillas si
arrastró el archivo

        try
        {
            contenido = File.ReadAllText(rutaArchivo);
            Console.WriteLine("Archivo cargado: " + contenido.Length + "
caracteres.");
            return true;
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("El archivo no existe. Revisa la ruta.");
            return false;
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo leer: " + ex.Message);
            return false;
        }
    }

    public void Estadisticas()
    {
        if (string.IsNullOrEmpty(contenido))
        {
            Console.WriteLine("Primero carga un archivo.");
            return;
        }

        // Líneas: contar saltos de línea (+1 si no termina en salto)
        int lineas = contenido.Split('\n').Length;

        // Palabras: separar por espacios y signos comunes
        char[] separadores = { ' ', '\n', '\r', '\t', '.', ',', ';', ':', '!', '?', '&',
'?', '!', '(', ')' };

```

```

        string[] palabras = contenido.Split(separadores,
StringSplitOptions.RemoveEmptyEntries);

        int caracteresSinEspacios = contenido.Count(c => !char.IsWhiteSpace(c));

        Console.WriteLine("--- Estadísticas ---");
        Console.WriteLine("Líneas: " + lineas);
        Console.WriteLine("Palabras: " + palabras.Length);
        Console.WriteLine("Caracteres (sin espacios): " + caracteresSinEspacios);

        // Top 5 palabras más frecuentes (el patrón "diccionario para contar" de la
unidad 7)
        Dictionary<string, int> conteo = new Dictionary<string, int>();
        foreach (string p in palabras)
        {
            string clave = p.ToLower();
            if (conteo.ContainsKey(clave))
            {
                conteo[clave]++;
            }
            else
            {
                conteo[clave] = 1;
            }
        }

        var top5 = conteo.OrderByDescending(par => par.Value).Take(5);
        Console.WriteLine("Palabras más frecuentes:");
        foreach (var par in top5)
        {
            Console.WriteLine("  " + par.Key + ": " + par.Value + " veces");
        }
    }

    public void Buscar()
    {
        if (string.IsNullOrEmpty(contenido))
        {
            Console.WriteLine("Primero carga un archivo.");
            return;
        }

        Console.Write("Texto a buscar: ");
        string busqueda = Console.ReadLine();
    }

```

```

        if (string.IsNullOrEmpty(busqueda))
        {
            Console.WriteLine("Escribe algo para buscar.");
            return;
        }

        // Contar ocurrencias sin importar mayúsculas
        int ocurrencias = 0;
        int posicion = 0;
        while ((posicion = contenido.IndexOf(busqueda, posicion,
StringComparison.OrdinalIgnoreCase)) != -1)
        {
            ocurrencias++;
            posicion += busqueda.Length; // avanzar para no contar la misma dos veces
        }

        Console.WriteLine("'" + busqueda + "' aparece " + ocurrencias + " veces.");

        // Mostrar el contexto de la primera aparición (30 caracteres antes y después)
        if (ocurrencias > 0)
        {
            int primera = contenido.IndexOf(busqueda,
StringComparison.OrdinalIgnoreCase);
            int inicio = Math.Max(0, primera - 30);
            int fin = Math.Min(contenido.Length, primera + busqueda.Length + 30);
            string contexto = contenido.Substring(inicio, fin - inicio).Replace("\n",
" ");

            Console.WriteLine("Contexto: ..." + contexto + "...");
        }
    }

    public void Reemplazar()
    {
        if (string.IsNullOrEmpty(contenido))
        {
            Console.WriteLine("Primero carga un archivo.");
            return;
        }

        Console.Write("Texto a reemplazar: ");
        string original = Console.ReadLine();
        Console.Write("Texto nuevo: ");
        string nuevo = Console.ReadLine();

        if (string.IsNullOrEmpty(original))

```

```

    {
        Console.WriteLine("El texto a reemplazar no puede estar vacío.");
        return;
    }

    // Contar cuántos se reemplazarán antes de hacerlo
    int cuantos = contenido.Split(new string[] { original },
StringSplitOptions.None).Length - 1;
    contenido = contenido.Replace(original, nuevo);
    Console.WriteLine("Se reemplazaron " + cuantos + " ocurrencias.");

    Console.Write("¿Guardar los cambios en el archivo? (s/n): ");
    if (Console.ReadLine().Trim().ToLower() == "s")
    {
        try
        {
            // Respaldo antes de modificar: profesional y paranoico, como debe ser
            string respaldo = rutaArchivo + ".bak";
            File.Copy(rutaArchivo, respaldo, overwrite: true);
            File.WriteAllText(rutaArchivo, contenido);
            Console.WriteLine("Guardado. Respaldo en: " + respaldo);
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo guardar: " + ex.Message);
        }
    }
}

}

class Program
{
    static void Main()
    {
        ProcesadorTexto proc = new ProcesadorTexto();

        if (!proc.CargarArchivo())
        {
            return; // si no se pudo cargar, no hay nada que procesar
        }

        string opcion = "";
        while (opcion != "4")
        {
            Console.WriteLine();

```

```

        Console.WriteLine("=== PROCESADOR DE TEXTO ===");
        Console.WriteLine("1. Estadísticas");
        Console.WriteLine("2. Buscar texto");
        Console.WriteLine("3. Buscar y reemplazar");
        Console.WriteLine("4. Salir");
        Console.Write("Opción: ");
        opcion = Console.ReadLine().Trim();

        if (opcion == "1") proc.Estadísticas();
        else if (opcion == "2") proc.Buscar();
        else if (opcion == "3") proc.Reemplazar();
        else if (opcion != "4") Console.WriteLine("Opción no válida.");
    }
}

```

Explicación de los trucos importantes:

- ``CargarArchivo`` hace ``Trim("")``: cuando arrastras un archivo a la consola de Windows, la ruta llega entre comillas; quitarlas evita el error tonto de "archivo no existe". Detalle de quien ya sufrió ese bug.
- ``Estadísticas`` parte el texto con un arreglo de separadores amplio (espacios, puntuación, saltos) para contar palabras reales, no pedazos con comas pegadas. ``char.IsWhiteSpace(c)`` dentro de un ``Count`` de LINQ cuenta caracteres no blancos: LINQ también trabaja sobre cadenas porque son enumerables de char.
- El top de palabras usa el patrón "diccionario para contar" con clave en minúsculas (para que "El" y "el" cuenten juntos), luego ``OrderByDescending`` + ``Take(5)``: la unidad 7 pagando dividendos en la unidad 8.
- ``Buscar`` usa ``IndexOf`` en un ciclo para contar ocurrencias insensibles a mayúsculas con ``StringComparison.OrdinalIgnoreCase``, avanzando la posición para no recontar. Y muestra el CONTEXTO de la primera aparición con ``Substring`` protegido por ``Math.Max`/`Math.Min``: nunca se sale del rango, sin importar dónde aparezca la palabra.
- ``Reemplazar`` cuenta primero con el truco de ``Split`` (partir por el texto buscado: el número de pedazos menos uno es el número de ocurrencias) y, antes de guardar, crea un RESPALDO ``.bak`` con ``File.Copy(..., overwrite: true)``. Modificar el archivo original sin respaldo es de valientes; los profesionales respaldan.
- Nota el ``overwrite: true`` con nombre de parámetro: sintaxis de argumentos nombrados, que hace el código más legible cuando un booleano no se explica solo.

Ejemplo 3: Registro de errores (log) con StreamWriter y using

Un ejemplo corto que combina excepciones, archivos y `using`: una bitácora donde queda registrado todo lo que falla, con fecha y detalle. Es el componente que todo programa serio debería tener:

```
using System;
using System.IO;

static class Bitacora
{
    private static string rutaLog;

    // Constructor estático: se ejecuta una vez al usar la clase por primera vez
    static Bitacora()
    {
        string carpeta = Path.Combine(
            Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData),
            "MiPrograma");
        Directory.CreateDirectory(carpeta);
        rutaLog = Path.Combine(carpeta, "errores.log");
    }

    public static void Registrar(string mensaje)
    {
        // using garantiza que el archivo se cierra aunque truee al escribir
        using (StreamWriter w = new StreamWriter(rutaLog, append: true))
        {
            w.WriteLine(DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss") + " | " +
mensaje);
        }
    }

    public static void RegistrarError(Exception ex)
    {
        using (StreamWriter w = new StreamWriter(rutaLog, append: true))
        {
            w.WriteLine(DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss") + " | ERROR: " +
ex.GetType().Name);
            w.WriteLine(" Mensaje: " + ex.Message);
            w.WriteLine(" Origen: " + ex.StackTrace);
            w.WriteLine(new string('-', 40));
        }
    }
}
```

```

class Program
{
    static void Main()
    {
        Bitacora.Registrar("Programa iniciado.");

        try
        {
            // ... aquí va la lógica del programa ...
            int.Parse("no es número"); // provocamos un error a propósito
        }
        catch (Exception ex)
        {
            Bitacora.RegistrarError(ex);
            Console.WriteLine("Hubo un problema, pero quedó registrado en la
bitácora.");
        }

        Bitacora.Registrar("Programa terminado.");
    }
}

```

Con esto, cada error queda guardado con fecha, tipo, mensaje y stack trace en un archivo dentro de la carpeta de datos de la aplicación. Cuando un usuario reporte "me salió un error", le pides ese archivo y tienes toda la información para arreglarlo. Esa es la diferencia entre adivinar y diagnosticar.

Práctica

Igual que antes: intenta primero sin mirar, provoca los errores a propósito para ver qué pasa, y luego compara. En esta unidad es ESPECIALMENTE importante que ejecutes cada ejercicio y pruebes los casos que truenan (escribe letras donde pide números, borra el archivo antes de leerlo). Ver el error con tus propios ojos es lo que te enseña a prevenirlo.

1. División segura: pide dos números enteros al usuario y muestra la división. Usa try/catch para atrapar `FormatException` (si no escribe números) y `DivideByZeroException` (si el divisor es cero), con un mensaje distinto para cada caso. Si todo sale bien, muestra el resultado con 2 decimales.

2. Excepción propia: crea una clase `EdadInvalidaException` que herede de `Exception`. Luego crea un método `RegistrarEdad(int edad)` que lance esa excepción si la edad es menor a 0 o mayor a 130. En el `Main`, pide la edad con `TryParse` y atrapa tu excepción mostrando un mensaje amable.
3. Lista de compras persistente: haz un programa que guarde una lista de compras en "compras.txt" (una línea por producto). Al iniciar, si el archivo existe, carga los productos; el usuario puede agregar productos y al salir se guardan automáticamente. Usa `File.Exists`, `ReadAllLines` y `WriteAllLines`.
4. Contador de vocales con `StringBuilder`: lee un texto del usuario (varias líneas hasta línea vacía), ármalo con `StringBuilder`, y luego cuenta cuántas vocales (a, e, i, o, u, sin importar mayúsculas) contiene. Muestra el total y el texto en mayúsculas usando interpolación en el mensaje final.
5. Formato de ticket: crea un método que reciba nombre del cliente, lista de (producto, precio) y genere un ticket con formato profesional: encabezado, cada producto con precio en formato de moneda alineado, total con 2 decimales, y fecha en formato largo. Guarda el ticket en un archivo con el nombre del cliente.
6. Validador de contraseñas: pide una contraseña y valida con métodos de cadena: mínimo 8 caracteres, al menos una mayúscula, al menos una minúscula y al menos un dígito. Usa `Length`, `char.IsUpper`, `char.IsLower` y `char.IsDigit` recorriendo la cadena. Muestra qué reglas fallaron (usa `StringBuilder` para armar el reporte de errores).
7. Copia de seguridad: escribe un método `Respalidar(string rutaOriginal)` que copie un archivo a la misma carpeta con la fecha de hoy en el nombre (ejemplo: "datos_2026-09-19.bak"). Debe: verificar que el original exista, armar la ruta con `Path.Combine` y `Path.GetDirectoryName`, atrapar `IOException` y devolver `true/false` según el resultado.
8. Diario personal: combina todo. Un programa con menú que permite: escribir una entrada (se guarda con fecha y hora al final de "diario.txt" usando `AppendAllText`), leer todo el diario, y buscar una palabra en el diario mostrando en qué líneas aparece (usa `Split` por líneas y `Contains` insensible a mayúsculas). Todo el acceso a archivo dentro de `try/catch`.

Soluciones

Solución 1: división segura con `catch` específicos. Cada error con su mensaje:

```

using System;

class Program
{
    static void Main()
    {
        try
        {
            Console.Write("Dividendo: ");
            int a = int.Parse(Console.ReadLine());
            Console.Write("Divisor: ");
            int b = int.Parse(Console.ReadLine());

            int resultado = a / b;
            Console.WriteLine($"Resultado: {a} / {b} = {(double)a / b:N2}");
        }
        catch (FormatException)
        {
            Console.WriteLine("Error: debes escribir números enteros, no texto.");
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Error: no se puede dividir entre cero.");
        }
    }
}

```

Nota dos cosas: los catch van de específico a general (si agregáramos `catch (Exception)` iría al final), y el resultado se muestra como double con formato `N2` para ver los decimales: la división entera `a / b` trunca, así que convertimos a double para el despliegue.

Solución 2: excepción propia. Definir el tipo, lanzarlo y atraparlo:

```

using System;

// Excepción propia: hereda de Exception y lleva la edad inválida como dato
class EdadInvalidaException : Exception
{
    public int EdadRecibida { get; }

    public EdadInvalidaException(string mensaje, int edad) : base(mensaje)
    {
        EdadRecibida = edad;
    }
}

```

```

class Program
{
    static void RegistrarEdad(int edad)
    {
        if (edad < 0 || edad > 130)
        {
            throw new EdadInvalidaException("La edad no parece realista.", edad);
        }
        Console.WriteLine($"Edad registrada: {edad} años.");
    }

    static void Main()
    {
        Console.Write("Escribe tu edad: ");
        int edad;

        if (!int.TryParse(Console.ReadLine(), out edad))
        {
            Console.WriteLine("Eso no es un número válido.");
            return;
        }

        try
        {
            RegistrarEdad(edad);
        }
        catch (EdadInvalidaException ex)
        {
            Console.WriteLine($"Error: {ex.Message} (recibido: {ex.EdadRecibida})");
        }
    }
}

```

El método `RegistrarEdad` valida y lanza; el Main convierte con `TryParse` (sin excepciones para entrada normal) y atrapa solo la excepción de negocio. Cada capa maneja lo que le toca: conversión en la entrada, reglas de negocio en el método.

Solución 3: lista de compras persistente. El archivo como memoria del programa:

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

class Program

```

```

{
    static void Main()
    {
        string ruta = "compras.txt";
        List<string> compras = new List<string>();

        // Al iniciar: cargar si existe
        try
        {
            if (File.Exists(ruta))
            {
                compras = File.ReadAllLines(ruta).ToList();
                Console.WriteLine($"Se cargaron {compras.Count} productos
guardados.");
            }
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo cargar la lista anterior: " + ex.Message);
        }

        string entrada = "";
        while (entrada.ToLower() != "fin")
        {
            Console.Write("Agrega un producto (o 'fin' para terminar): ");
            entrada = Console.ReadLine().Trim();

            if (!string.IsNullOrEmpty(entrada) && entrada.ToLower() != "fin")
            {
                compras.Add(entrada);
            }
        }

        // Al salir: guardar todo
        try
        {
            File.WriteAllLines(ruta, compras);
            Console.WriteLine($"Lista guardada con {compras.Count} productos.");
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo guardar la lista: " + ex.Message);
            Console.WriteLine("Tus productos de esta sesión:");
            foreach (string p in compras)
            {

```

```

        Console.WriteLine("- " + p);
    }
}
}
}

```

Fíjate en el detalle profesional del catch al guardar: si no se puede escribir el archivo, al menos muestra los productos en pantalla para que el usuario los copie a mano. Nunca dejes que el trabajo del usuario se pierda en silencio.

Solución 4: contador de vocales. StringBuilder para armar, Contains para verificar:

```

using System;
using System.Text;

class Program
{
    static void Main()
    {
        Console.WriteLine("Escribe tu texto (línea vacía para terminar:");
        StringBuilder sb = new StringBuilder();
        string linea;
        while (!string.IsNullOrEmpty(linea = Console.ReadLine()))
        {
            sb.AppendLine(linea);
        }

        string texto = sb.ToString();
        string vocales = "aeiouAEIOU";
        int contador = 0;

        foreach (char c in texto)
        {
            if (vocales.Contains(c))
            {
                contador++;
            }
        }

        Console.WriteLine($"Tu texto tiene {contador} vocales.");
        Console.WriteLine($"En mayúsculas se ve así:\n{texto.ToUpper()}");
    }
}

```

El truco `vocales.Contains(c)`: como `string` es una secuencia de caracteres, `Contains` sirve para preguntar si un char está en el conjunto de vocales. Simple y legible. Y la interpolación con `\n` adentro muestra que puedes incluir saltos de línea en el mensaje.

Solución 5: ticket con formato. Interpolación + formatos + archivo:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

class Program
{
    static void GenerarTicket(string cliente, List<(string Producto, decimal Precio)>
items)
    {
        StringBuilder ticket = new StringBuilder();
        ticket.AppendLine("=====");
        ticket.AppendLine("          TIENDA LUCAS EGA          ");
        ticket.AppendLine($"Fecha: {DateTime.Now:D}");
        ticket.AppendLine($"Cliente: {cliente}");
        ticket.AppendLine("=====");

        decimal total = 0;
        foreach (var item in items)
        {
            // Alineación: el producto a la izquierda (20 espacios), el precio a la
derecha (10)
            ticket.AppendLine($"{{item.Producto,-20}} {{item.Precio,10:C}}");
            total += item.Precio;
        }

        ticket.AppendLine("=====");
        ticket.AppendLine($"TOTAL:{{total,26:C}}");
        ticket.AppendLine("¡Gracias por su compra!");

        string nombreArchivo = cliente.Replace(" ", "_") + "_ticket.txt";
        File.WriteAllText(nombreArchivo, ticket.ToString());
        Console.WriteLine(ticket.ToString());
        Console.WriteLine("Ticket guardado en: " + nombreArchivo);
    }

    static void Main()
    {
        var compra = new List<(string Producto, decimal Precio)>
```



```

    {
        ("Laptop", 15000m),
        ("Mouse", 350m),
        ("Teclado", 800m),
    };
    GenerarTicket("Ana García", compra);
}
}

```

Lo nuevo aquí es la ALINEACIÓN en la interpolación: `{item.Producto,-20}` reserva 20 espacios alineando a la izquierda (el menos indica izquierda), y `{item.Precio,10:C}` reserva 10 alineando a la derecha con formato de moneda. Así las columnas quedan derechitas como en un ticket real. Es el mismo truco que usan los reportes de consola profesionales.

Solución 6: validador de contraseñas. Recorrer la cadena con `char.IsUpper/IsLower/IsDigit`:

```

using System;
using System.Text;

class Program
{
    static void Main()
    {
        Console.Write("Crea tu contraseña: ");
        string clave = Console.ReadLine();

        StringBuilder errores = new StringBuilder();
        bool tieneMayus = false, tieneMinus = false, tieneDigito = false;

        if (string.IsNullOrEmpty(clave) || clave.Length < 8)
        {
            errores.AppendLine("- Mínimo 8 caracteres.");
        }

        if (!string.IsNullOrEmpty(clave))
        {
            foreach (char c in clave)
            {
                if (char.IsUpper(c)) tieneMayus = true;
                if (char.IsLower(c)) tieneMinus = true;
                if (char.IsDigit(c)) tieneDigito = true;
            }
        }

        if (!tieneMayus) errores.AppendLine("- Al menos una mayúscula.");
        if (!tieneMinus) errores.AppendLine("- Al menos una minúscula.");
        if (!tieneDigito) errores.AppendLine("- Al menos un dígito.");

        if (errores.Length == 0)
        {
            Console.WriteLine("Contraseña válida. ¡Bienvenido!");
        }
        else
        {
            Console.WriteLine("La contraseña no cumple:");
            Console.Write(errores.ToString());
        }
    }
}

```

En vez de decir "contraseña inválida" a secas, el programa dice EXACTAMENTE qué reglas fallaron. Esa es la diferencia entre un programa frustrante y uno que ayuda. ``StringBuilder`` arma el reporte y ``errores.Length == 0`` pregunta si hubo algún error.

Solución 7: copia de seguridad con fecha. Rutas armadas correctamente:

```

using System;
using System.IO;

class Program
{
    static bool Respaldar(string rutaOriginal)
    {
        try
        {
            if (!File.Exists(rutaOriginal))
            {
                Console.WriteLine("El archivo original no existe: " +
rutaOriginal);
                return false;
            }

            string carpeta = Path.GetDirectoryName(rutaOriginal);
            string nombreSinExt = Path.GetFileNameWithoutExtension(rutaOriginal);
            string fecha = DateTime.Now.ToString("yyyy-MM-dd");
            string nombreRespaldo = nombreSinExt + "_" + fecha + ".bak";
            string rutaRespaldo = Path.Combine(carpeta, nombreRespaldo);

            File.Copy(rutaOriginal, rutaRespaldo, overwrite: true);
            Console.WriteLine("Respaldo creado: " + rutaRespaldo);
            return true;
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo crear el respaldo: " + ex.Message);
            return false;
        }
    }

    static void Main()
    {
        // Para probar: primero crea el archivo
        File.WriteAllText("datos.txt", "información importante");
        bool ok = Respaldar(Path.GetFullPath("datos.txt"));
        Console.WriteLine("¿Funcionó? " + ok);
    }
}

```

`Path.GetDirectoryName` extrae la carpeta, `GetFileNameWithoutExtension` el nombre limpio, y la fecha en formato `yyyy-MM-dd` hace que los respaldos se ordenen cronológicamente

solos. El método devuelve bool para que quien lo llama sepa si funcionó: nada de excepciones para resultados esperables.

Solución 8: diario personal. El examen final de la unidad:

```
using System;
using System.IO;
using System.Linq;

class Program
{
    static string ruta = Path.Combine(
        Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments),
        "diario.txt");

    static void EscribirEntrada()
    {
        Console.WriteLine("Escribe tu entrada (línea vacía para terminar):");
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        sb.AppendLine("=== " + DateTime.Now.ToString("yyyy-MM-dd HH:mm") + " ===");

        string linea;
        while (!string.IsNullOrEmpty(linea = Console.ReadLine()))
        {
            sb.AppendLine(linea);
        }
        sb.AppendLine(); // línea en blanco separando entradas

        try
        {
            File.AppendAllText(ruta, sb.ToString());
            Console.WriteLine("Entrada guardada en tu diario.");
        }
        catch (IOException ex)
        {
            Console.WriteLine("No se pudo guardar: " + ex.Message);
        }
    }

    static void LeerDiario()
    {
        try
        {
            if (!File.Exists(ruta))
            {
                Console.WriteLine("Tu diario está vacío. ¡Escribe tu primera
```

```

entrada!");
        return;
    }
    Console.WriteLine(File.ReadAllText(ruta));
}
catch (IOException ex)
{
    Console.WriteLine("No se pudo leer: " + ex.Message);
}
}

static void BuscarEnDiario()
{
    Console.Write("Palabra a buscar: ");
    string palabra = Console.ReadLine();

    if (string.IsNullOrEmpty(palabra))
    {
        Console.WriteLine("Escribe una palabra para buscar.");
        return;
    }

    try
    {
        if (!File.Exists(ruta))
        {
            Console.WriteLine("Tu diario está vacío.");
            return;
        }

        string[] lineas = File.ReadAllLines(ruta);
        var encontradas = lineas
            .Select((texto, indice) => new { texto, indice })
            .Where(x => x.texto.Contains(palabra,
StringComparison.OrdinalIgnoreCase))
            .ToList();

        if (encontradas.Any())
        {
            Console.WriteLine($"'{palabra}' aparece en {encontradas.Count}
líneas:");
            foreach (var e in encontradas)
            {
                Console.WriteLine($"  Línea {e.indice + 1}: {e.texto.Trim()}");
            }
        }
    }
}

```

```

        }
        else
        {
            Console.WriteLine($"'{palabra}' no aparece en tu diario.");
        }
    }
    catch (IOException ex)
    {
        Console.WriteLine("No se pudo buscar: " + ex.Message);
    }
}

static void Main()
{
    string opcion = "";
    while (opcion != "4")
    {
        Console.WriteLine();
        Console.WriteLine("=== MI DIARIO ===");
        Console.WriteLine("1. Escribir entrada");
        Console.WriteLine("2. Leer diario");
        Console.WriteLine("3. Buscar palabra");
        Console.WriteLine("4. Salir");
        Console.Write("Opción: ");
        opcion = Console.ReadLine().Trim();

        if (opcion == "1") EscribirEntrada();
        else if (opcion == "2") LeerDiario();
        else if (opcion == "3") BuscarEnDiario();
        else if (opcion != "4") Console.WriteLine("Opción no válida.");
    }
    Console.WriteLine(";Hasta mañana!");
}
}

```

Este programa junta TODA la unidad: `AppendAllText` para agregar sin borrar, `ReadAllLines` para procesar por líneas, excepciones específicas con mensajes amables, `StringBuilder` para armar la entrada, interpolación para los mensajes, `Contains` insensible a mayúsculas para buscar, y LINQ con `Select` de dos parámetros (texto e índice) para saber el número de línea. Si entendiste este programa completo, entendiste la unidad.

Resumen de la unidad

- Las excepciones son errores en tiempo de ejecución: el código compila bien pero algo falla mientras corre (entrada inválida, archivo inexistente, división entre cero). Si nadie las atiende, el programa se cierra de golpe.
- ``try`` envuelve el código riesgoso; ``catch`` atiende cada tipo de excepción (los específicos primero, el genérico al final); ``finally`` se ejecuta siempre y sirve para limpieza. Puedes tener `try/catch` sin `finally` o `try/finally` sin `catch`.
- Memoriza las comunes: `FormatException` (texto con formato inválido), `NullReferenceException` (usar algo null), `DivideByZeroException`, `IndexOutOfRangeException`, `KeyNotFoundException`, `InvalidOperationException`, `ArgumentException`, `IOException` y `FileNotFoundException`.
- ``throw`` lanza excepciones desde tus métodos cuando detectan algo inválido; crea tus propios tipos heredando de `Exception` cuando el error sea específico de tu negocio. Re-lanza con ``throw;`` (solo) para conservar el `StackTrace`.
- Atrapa cuando puedas HACER ALGO ÚTIL (reintentar, valor por defecto, mensaje amable) o en la FRONTERA del programa con registro. NO atrapes errores de programación (se arreglan, no se atrapan) y NUNCA dejes un `catch` vacío.
- Para convertir texto del usuario usa ``TryParse`` (devuelve true/false) en vez de ``Parse`` con `try/catch`. Es el patrón "intentar sin tronar", igual que `TryGetValue`.
- ``File.WriteAllText`/`ReadAllText`` escriben y leen archivos completos; ``AppendAllText`` agrega sin borrar; ``WriteAllLines`/`ReadAllLines`` trabajan con arreglos de líneas. Verifica ``Exists`` antes de leer y recuerda que escribir SOBREScribe sin preguntar.
- Arma rutas portables: relativas o con ``Path.Combine``, carpetas del sistema con ``Environment.SpecialFolder``, y ``Directory.CreateDirectory`` antes de escribir en subcarpetas. Nunca hardcodees rutas absolutas de tu máquina.
- ``StreamReader`/`StreamWriter`` leen y escriben por partes (ideales para archivos grandes); el patrón ``while ((linea = lector.ReadLine()) != null)`` lee hasta el final.
- ``using`` cierra y libera recursos automáticamente al salir del bloque, pase lo que pase. Úsalo siempre con archivos, conexiones y todo lo que implemente `IDisposable`. No usarlo deja archivos bloqueados.
- Los string son inmutables: cada "modificación" crea uno nuevo. Para concatenar en ciclos o armar textos grandes usa ``StringBuilder`` (`Append`, `AppendLine`, `ToString`); para lo demás, interpolación ``$"...``.

- Interpolación con formato: `{precio:C}` moneda, `{n:N2}` número con 2 decimales, `{p:P0}` porcentaje, `{fecha:d}/{fecha:D}` fechas, y alineación `{texto,-20}` / `{num,10}` para columnas derechitas.
- Caja de herramientas del texto: Split (partir), Join (unir, es estático), Contains/StartsWith/EndsWith (preguntar, con StringComparison.OrdinalIgnoreCase para ignorar mayúsculas), Replace, Substring (verifica rangos), Trim (limpia espacios), ToUpper/ToLower, IndexOf (-1 si no encuentra).
- Valida la entrada con `string.IsNullOrEmpty` antes de procesarla y haz `Trim()` a lo que leas del usuario. Los métodos de string devuelven cadenas NUEVAS: si no asignas el resultado, el cambio se pierde.
- Buenas prácticas: mensajes de error que el usuario entienda, respaldos antes de modificar archivos, confirmación antes de borrar, registro de errores con fecha en una bitácora, y programar a la defensiva porque los datos de afuera siempre vienen chuecos.

UNIDAD 9: Asincronía, delegados y genéricos

Bienvenido a la Unidad 9 del Curso Completo de C#. Si llegaste hasta aquí, ya dominas las bases del lenguaje: variables, decisiones, ciclos, métodos, colecciones, programación orientada a objetos y el manejo de errores. Ahora es momento de subir de nivel y aprender tres temas que separan a un programador que "hace programas que funcionan" de un programador que "hace programas que funcionan bien en el mundo real": la asincronía con `async` y `await`, los delegados y eventos, y los genéricos.

Estos tres temas tienen algo en común: aparecen en todos los trabajos de C#. Cuando veas código profesional, en una empresa, en un proyecto de código abierto o en una entrevista de trabajo, vas a encontrar `Task`, `await`, `Func`, `Action`, eventos y `List` con genéricos por todas partes. No son adornos del lenguaje: son las herramientas que permiten construir aplicaciones rápidas, responsivas y reutilizables.

Empecemos por la asincronía. Imagina que entras a una aplicación de escritorio, haces clic en un botón que dice "Descargar reporte" y toda la ventana se congela. No puedes moverla, no puedes cerrarla, el cursor muestra el círculo de "cargando" durante veinte segundos. Eso es un programa síncrono mal diseñado: mientras espera que la descarga termine, el programa no puede hacer nada más. En cambio, un programa asíncrono bien hecho te deja seguir trabajando, moviendo la ventana, escribiendo en otros campos, mientras la descarga avanza en segundo plano. La diferencia entre uno y otro es exactamente lo que vas a aprender en la primera parte de esta unidad: las palabras clave `async` y `await`, el tipo `Task`, y métodos como `Task.Run`, `Task.Delay` y `Task.WhenAll`.

La analogía clásica para entender la asincronía es la del mesero en un restaurante. Un mesero síncrono atiende una mesa, toma la orden, se queda parado junto a la cocina esperando a que el cocinero termine el platillo, lo lleva a la mesa y recién entonces va a atender a la siguiente mesa. Con cinco mesas llenas, el servicio es un desastre: todos esperan. Un mesero asíncrono hace lo inteligente: toma la orden de la mesa uno, la pasa a la cocina, y mientras el cocinero trabaja, va a tomar la orden de la mesa dos, luego la de la mesa tres, y así sucesivamente. Cuando un platillo está listo, lo recoge y lo sirve. El mesero nunca se queda parado esperando: siempre está haciendo algo útil. Eso es `async` y `await` en C#: tu programa toma una tarea que tarda (leer un archivo, descargar datos, consultar una base de datos), la "pasa a la cocina" y mientras tanto sigue atendiendo otras cosas. Cuando la tarea termina, el programa retoma justo donde se había quedado.

¿Por qué importa esto en la vida real? Por dos razones enormes. La primera es la experiencia del usuario: nadie quiere usar una aplicación que se congela. La segunda es la escalabilidad

de los servidores. Un servidor web moderno atiende miles de peticiones al mismo tiempo. Si cada petición bloqueara un hilo del procesador mientras espera a la base de datos, el servidor necesitaría miles de hilos, consumiría una cantidad brutal de memoria y se caería. Con `async` y `await`, un solo hilo puede atender cientos de peticiones: cuando una petición espera a la base de datos, el hilo se libera y atiende otra; cuando llegan los datos, la petición original continúa. Es la diferencia entre un servidor que atiende a 100 usuarios y uno que atiende a 100,000 con el mismo hardware.

La segunda parte de esta unidad trata sobre delegados y eventos. Un delegado es, en esencia, una variable que guarda un método en lugar de un valor. Piensa en el control remoto de la televisión: el botón de "encendido" no sabe cómo encender la tele por dentro, solo sabe que cuando lo presionas, algo pasa. Un delegado es igual: guarda una referencia a un método y permite llamarlo después, sin saber de antemano qué método será. Esto permite patrones muy poderosos, como pasar comportamiento como parámetro: "ordena esta lista, y aquí te dejo el método que decide qué va antes que qué". C# trae delegados genéricos listos para usar: `Func<T>` para métodos que devuelven un valor, `Action<T>` para métodos que no devuelven nada, y `Predicate<T>` para métodos que devuelven verdadero o falso. Y con las expresiones lambda (la flechita `=>`) puedes escribir esos métodos en una sola línea, sin declarar una función completa.

Los eventos construyen sobre los delegados el patrón publicador-suscriptor: un objeto "publica" que algo ocurrió (un botón fue presionado, un archivo terminó de descargarse, un temporizador llegó a cero) y otros objetos se "suscriben" para reaccionar. Es el corazón de las interfaces gráficas: cuando haces clic en un botón, se dispara un evento `Click` y el método que escribiste se ejecuta. Sin eventos, tendrías que estar preguntando todo el tiempo "¿ya hicieron clic? ¿ya hicieron clic?", lo cual es terrible. Con eventos, el programa avisa solo cuando pasa algo.

La tercera parte cubre los genéricos. Ya los has usado sin darte cuenta: `List<string>`, `Dictionary<string, int>` son genéricos. Un genérico es una clase o un método que trabaja con "un tipo que se decide después". En lugar de escribir una clase `ListaDeEnteros`, otra `ListaDeCadenas` y otra `ListaDeProductos`, escribes una sola clase `Lista<T>` y la `T` se reemplaza por el tipo que necesites al usarla. Los genéricos te dan seguridad de tipos en tiempo de compilación (el compilador te avisa si metes un texto donde van números), mejor rendimiento (sin conversiones de ida y vuelta entre `object` y el tipo real) y código reutilizable. Vas a aprender a crear tus propias clases y métodos genéricos, y a usar restricciones con `where T` para decirle al compilador qué debe cumplir el tipo `T` (por ejemplo, que tenga un constructor sin parámetros, o que implemente cierta interfaz). También entenderás por qué `List<T>` es superior al viejo `ArrayList` de los primeros días de .NET: `ArrayList` guarda todo

como `object`, lo que obliga a conversiones constantes y permite meter cualquier cosa en la lista, con errores que solo aparecen al ejecutar el programa.

Al terminar esta unidad podrás escribir métodos asíncronos que no congelan la interfaz, lanzar varias tareas en paralelo y esperarlas todas con `Task.WhenAll`, evitar los errores clásicos como `async void` o olvidar un `await`, usar `Func`, `Action` y lambdas para pasar comportamiento como datos, crear tus propios eventos siguiendo el patrón publicador-suscriptor, y diseñar clases y métodos genéricos con restricciones. Son habilidades de programador profesional, y las vas a practicar con ejemplos comentados línea por línea y ejercicios con solución.

Explicación detallada

El problema que resuelve la asincronía

Todo programa ejecuta instrucciones en un orden: primero esta línea, luego la siguiente. Cuando una instrucción tarda mucho (leer un archivo del disco, descargar una página de internet, esperar la respuesta de una base de datos), el programa se queda esperando. En un programa de consola sencillo eso no se nota mucho, pero en una aplicación con interfaz gráfica el efecto es devastador: el hilo principal, el que dibuja la ventana y responde a tus clics, queda bloqueado y la aplicación "se congela". Windows incluso te muestra el molesto mensaje de "la aplicación no responde".

Históricamente, la solución era crear hilos manualmente con la clase `Thread`. Funciona, pero es complicado: tienes que coordinar los hilos, evitar que dos hilos modifiquen el mismo dato al mismo tiempo, y cada hilo consume alrededor de 1 MB de memoria solo por existir. Para un servidor que atiende miles de peticiones, crear miles de hilos es inviable.

La asincronía con `async` y `await` es la respuesta moderna de C#. La idea clave es esta: cuando el programa llega a una operación que tarda, en lugar de bloquearse esperando, "devuelve el control" para que se pueda hacer otro trabajo, y cuando la operación termina, el programa continúa exactamente donde se había quedado, como si nada hubiera pasado. El código se sigue leyendo de arriba hacia abajo, casi igual que el código síncrono, pero por debajo el compilador hace la magia de pausar y reanudar.

Es importante entender una distinción: asíncrono no significa paralelo, aunque a veces se usan juntos. Asíncrono significa "no me bloqueo mientras espero". Paralelo significa "hago varias cosas al mismo tiempo en varios hilos". Puedes tener código asíncrono que corre en un solo hilo (muy común en interfaces gráficas y servidores web) y puedes combinarlo con paralelismo real cuando lo necesites con Task.Run. No los confundas: la asincronía es sobre esperar sin bloquear; el paralelismo es sobre ejecutar al mismo tiempo.

async, await y Task: las tres piezas

En C#, la asincronía se construye con tres piezas que siempre van juntas. La primera es el tipo Task (o Task<T> cuando la operación devuelve un valor). Un Task representa "una operación que está en curso o que terminará en el futuro". Es como un vale o un ticket: cuando pides tu comida en el restaurante te dan un número de ticket; el ticket no es la comida, es la promesa de que la comida llegará. Task es igual: no es el resultado, es la promesa de un resultado futuro.

La segunda pieza es la palabra clave async. Se coloca en la declaración del método para indicar "este método contiene operaciones asíncronas". Por sí sola no hace que el método sea asíncrono: solo le dice al compilador que dentro de ese método puede aparecer await. Un método async normalmente devuelve Task, Task<T> o, en casos especiales que veremos, void.

La tercera pieza es await. Se coloca delante de una operación que devuelve un Task y significa: "espera a que esta operación termine, pero sin bloquear el hilo; mientras tanto, deja que se haga otro trabajo, y cuando termine, continúa con la siguiente línea". El await es el mesero que deja la orden en la cocina y va a atender otras mesas: no se queda parado.

Veamos el ejemplo más simple posible:

```

using System;
using System.Threading.Tasks;

class Programa
{
    // Método async: lleva la palabra async y devuelve Task
    static async Task SaludarAsync()
    {
        Console.WriteLine("Empezando el saludo...");
        // await: espera 2 segundos SIN bloquear el hilo
        await Task.Delay(2000);
        Console.WriteLine("¡Hola después de 2 segundos!");
    }

    static async Task Main(string[] args)
    {
        Console.WriteLine("Antes de llamar al método");
        // await: espera a que SaludarAsync termine
        await SaludarAsync();
        Console.WriteLine("Después del método");
    }
}

```

El orden de los mensajes será: "Antes de llamar al método", "Empezando el saludo...", (pausa de 2 segundos sin bloquear), "¡Hola después de 2 segundos!", "Después del método". Observa que Main también es async: en C# moderno, el punto de entrada Main puede ser asíncrono, lo cual es muy conveniente.

Si el método necesita devolver un valor, usas Task<T> donde T es el tipo del valor:

```

using System;
using System.Threading.Tasks;

class Programa
{
    // Devuelve Task<int>: promete un entero en el futuro
    static async Task<int> SumarAsync(int a, int b)
    {
        await Task.Delay(1000); // Simula trabajo que tarda
        return a + b; // El return entrega el valor dentro del Task
    }

    static async Task Main(string[] args)
    {
        // await desempaqueta el Task<int> y nos da el int directamente
        int resultado = await SumarAsync(5, 3);
        Console.WriteLine("Resultado: " + resultado); // Resultado: 8
    }
}

```

Fíjate en el detalle importante: dentro del método escribes "return a + b" como si fuera un método normal que devuelve int, pero la firma dice Task<int>. Y quien llama al método usa await para obtener el int "desempaquetado". El compilador se encarga de envolver y desenvolver el valor. Esta es una de las bellezas del modelo async/await: el código asíncrono se lee casi igual que el síncrono.

Task.Run: trabajo pesado en segundo plano

Hay dos tipos de espera: esperar a que algo externo termine (una descarga, una lectura de disco, una consulta a base de datos) y hacer cálculos pesados que usan mucho procesador. Para el primer caso, las operaciones asíncronas naturales (como las de red o archivos) ya existen en .NET y solo hay que esperarlas con await. Para el segundo caso, cuando tienes un cálculo que tarda varios segundos usando el CPU al 100%, necesitas Task.Run.

Task.Run toma un método (normalmente una lambda) y lo ejecuta en un hilo del pool de hilos, devolviendo un Task que puedes esperar con await. Mientras el cálculo corre en ese hilo separado, tu hilo principal queda libre:

```

using System;
using System.Threading.Tasks;

class Programa
{
    // Método síncrono con cálculo pesado (NO lleva async)
    static long ContarPrimos(long limite)
    {
        long total = 0;
        for (long i = 2; i <= limite; i++)
        {
            bool esPrimo = true;
            for (long j = 2; j * j <= i; j++)
            {
                if (i % j == 0) { esPrimo = false; break; }
            }
            if (esPrimo) total++;
        }
        return total;
    }

    static async Task Main(string[] args)
    {
        Console.WriteLine("Empezando el cálculo pesado...");
        // Task.Run: ejecuta el cálculo en otro hilo, sin bloquear este
        long primos = await Task.Run(() => ContarPrimos(2000000));
        Console.WriteLine("Primos encontrados: " + primos);
        Console.WriteLine("El programa siguió respondiendo mientras calculaba.");
    }
}

```

La regla de oro es: usa `await` con operaciones naturalmente asíncronas (red, archivos, base de datos) directamente, y usa `Task.Run` solo para trabajo pesado de CPU que quieres sacar del hilo principal. Un error común de principiantes es envolver todo en `Task.Run` "por si acaso", incluso operaciones que ya tienen versión asíncrona. Eso desperdicia hilos del pool sin necesidad. Por ejemplo, para descargar una página web no haces `Task.Run(() => Descargar())`, sino que usas `await httpClient.GetStringAsync(url)`, que es asíncrono de verdad y no ocupa ningún hilo mientras espera la respuesta de la red.

Task.Delay y Task.WhenAll: esperar y combinar

Task.Delay es el temporizador asíncrono: espera una cantidad de tiempo sin bloquear el hilo. Es el reemplazo asíncrono de Thread.Sleep. Mientras Thread.Sleep congela el hilo por completo, Task.Delay lo libera para hacer otro trabajo. Lo usarás mucho para simular operaciones lentas en ejemplos y pruebas, para reintentos con pausa ("si falla, espera 2 segundos e intenta de nuevo") y para temporizadores.

Task.WhenAll es una de las herramientas más poderosas: recibe varias tareas y devuelve un Task que termina cuando TODAS terminan. Esto te permite lanzar operaciones en paralelo y esperarlas juntas, en lugar de esperarlas una por una. La diferencia de tiempo puede ser enorme:

```

using System;
using System.Threading.Tasks;

class Programa
{
    static async Task<string> DescargarAsync(string sitio)
    {
        Console.WriteLine("Descargando " + sitio + "...");
        await Task.Delay(2000); // Simula una descarga de 2 segundos
        return "Contenido de " + sitio;
    }

    static async Task Main(string[] args)
    {
        // FORMA LENTA: una por una, 2 + 2 + 2 = 6 segundos
        // string a = await DescargarAsync("sitio1");
        // string b = await DescargarAsync("sitio2");
        // string c = await DescargarAsync("sitio3");

        // FORMA RÁPIDA: las tres a la vez, total 2 segundos
        Task<string> t1 = DescargarAsync("sitio1");
        Task<string> t2 = DescargarAsync("sitio2");
        Task<string> t3 = DescargarAsync("sitio3");

        // WhenAll espera a que las tres terminen
        string[] resultados = await Task.WhenAll(t1, t2, t3);

        foreach (string r in resultados)
        {
            Console.WriteLine(r);
        }
    }
}

```

Fíjate en el truco: las tareas se INICIAN al llamar al método (sin await todavía), y luego un solo await con WhenAll las espera a todas. Como las tres descargas corren "al mismo tiempo" (cada una esperando su respuesta de red sin bloquear), el total es 2 segundos en lugar de 6. En un programa real que consulta tres APIs, esta técnica puede reducir el tiempo de espera a una tercera parte.

Existe también `Task.WhenAny`, que termina cuando la PRIMERA de las tareas termina. Sirve para escenarios como "consulta tres servidores y quédate con el que responda primero" o para implementar tiempos de espera máximos (timeout): lanzas tu operación y una tarea de `Task.Delay` como límite; si gana el Delay, cancelas.

Errores comunes en asincronía (y cómo evitarlos)

La asincronía tiene trampas famosas. Conocerlas de antemano te ahorra horas de depuración.

El primer error es `async void`. Un método `async` debería devolver `Task` o `Task<T>` casi siempre. La única excepción legítima es un manejador de eventos de interfaz gráfica (como un clic de botón), porque la firma del evento lo exige. ¿Por qué es malo `async void`? Porque quien llama al método no recibe ningún `Task`, así que no puede esperarlo con `await` ni puede capturar sus excepciones. Si un método `async void` lanza una excepción, esa excepción no tiene a dónde ir y puede tumbar toda la aplicación. Además, el llamador no sabe cuándo terminó el método. Regla simple: `async void` solo para eventos; en todo lo demás, devuelve `Task`.

```

using System;
using System.Threading.Tasks;

class Programa
{
    // MAL: async void, nadie puede esperar este método ni atrapar sus errores
    static async void MetodoMal()
    {
        await Task.Delay(1000);
        throw new Exception("Error que nadie puede atrapar bien");
    }

    // BIEN: devuelve Task, se puede esperar y sus errores se atrapan normal
    static async Task MetodoBienAsync()
    {
        await Task.Delay(1000);
        Console.WriteLine("Terminé, y si fallo, el await lanza la excepción.");
    }

    static async Task Main(string[] args)
    {
        try
        {
            await MetodoBienAsync(); // Los errores llegan aquí al try/catch
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error atrapado: " + ex.Message);
        }
    }
}

```

El segundo error es olvidar el `await`. Si llamas a un método `async` sin `await` (y sin guardar el `Task`), la operación arranca pero tu código sigue adelante sin esperarla. A veces eso es intencional ("fuego y olvida"), pero casi siempre es un descuido: el método no ha terminado cuando usas su resultado, o las excepciones se pierden. El compilador te avisa con una advertencia (warning CS4014) cuando detecta una llamada sin `await`: no la ignores. Si de verdad quieres lanzar y no esperar, guarda el `Task` en una variable para dejar clara la intención, o documenta por qué.

El tercer error clásico es el `deadlock` (abrazo mortal) al mezclar código síncrono y asíncrono. Ocurre cuando bloqueas un hilo esperando a que termine una tarea asíncrona con `.Result` o `.Wait()`, y esa tarea necesita volver a ese mismo hilo para continuar. El hilo espera a la tarea,

la tarea espera al hilo: nadie avanza y el programa se congela para siempre. Esto era muy común en aplicaciones de escritorio antiguas. La solución moderna es "async all the way": si una operación es asíncrona, toda la cadena de llamadas hacia arriba debe ser asíncrona, usando `await` en cada nivel, sin `.Result` ni `.Wait()` en el medio. En código nuevo, evita `.Result` y `.Wait()` por completo; si sientes la tentación de usarlos, es señal de que debes hacer `async` el método que llama.

El cuarto error es no pensar en las excepciones. En código asíncrono, las excepciones se lanzan en el `await`, así que el `try/catch` funciona de forma natural alrededor del `await`. Pero si usas `Task.WhenAll` con varias tareas y dos de ellas fallan, solo verás la primera excepción al hacer `await`; las demás quedan guardadas en la propiedad `Exception` del `Task` como `AggregateException`. Para escenarios donde necesitas todos los errores, inspecciona cada tarea individualmente después del `WhenAll`.

El quinto error es creer que `async` hace el código más rápido por arte de magia. La asincronía no acelera una operación individual: descargar un archivo tarda lo mismo con o sin `await`. Lo que mejora es el uso de recursos y la capacidad de hacer varias cosas a la vez. Si tu programa hace una sola cosa de principio a fin, `async` no lo hará más rápido; si atiende a muchos usuarios o hace muchas esperas, la diferencia es abismal.

Buenas prácticas de asincronía

La comunidad de C# ha destilado estas prácticas con años de experiencia. Primera: "async all the way". Desde el punto de entrada hasta la operación que espera, toda la cadena debe ser `async` con `await`. No rompas la cadena con `.Result`, `.Wait()` o `.GetAwaiter().GetResult()` en medio, porque ahí nacen los deadlocks.

Segunda: nombra tus métodos asíncronos con el sufijo `Async`. Es una convención universal en .NET: `DescargarDatosAsync`, `GuardarArchivoAsync`, `ObtenerUsuarioAsync`. Cuando alguien lee tu código, el sufijo le dice al instante "este método devuelve un `Task` y debo esperarlo con `await`". Los métodos de .NET ya siguen esta convención (`ReadAsync`, `WriteAsync`, `GetStringAsync`), así que tu código se verá consistente con la plataforma.

Tercera: evita `async void` excepto en manejadores de eventos. Ya vimos por qué.

Cuarta: para operaciones de entrada/salida (red, disco, base de datos), usa las versiones asíncronas que ya trae .NET en lugar de envolver versiones síncronas en `Task.Run`. `HttpClient`, `File.ReadAllTextAsync`, `StreamReader.ReadToEndAsync`, los métodos de `Entity Framework` como `ToListAsync`: todos existen para no desperdiciar hilos esperando.

Quinta: considera la cancelación con `CancellationToken` en operaciones largas. Un token de cancelación permite que el usuario cancele una descarga o que un servidor aborte trabajo innecesario. Es un tema un poco más avanzado, pero vale la pena que sepas que existe: los métodos asíncronos de .NET suelen aceptar un `CancellationToken` como último parámetro.

Sexta: no uses `Task.Delay` como "parche" para esperar a que algo termine ("duermo 2 segundos esperando que el archivo aparezca"). Eso es frágil: a veces 2 segundos no bastan y a veces sobran. Mejor espera de verdad a la operación con `await`, o usa primitivas de sincronización diseñadas para eso.

Delegados: variables que guardan métodos

Cambia de chip ahora: pasamos de la asincronía a los delegados. Un delegado es un tipo que representa un método: guarda su referencia para poder llamarlo después. La analogía del control remoto es perfecta: cuando el fabricante diseña el botón de encendido del control, no sabe qué televisor vas a comprar; solo sabe que el botón, al presionarse, debe "llamar" a la función de encendido del televisor que conectes. El delegado es el botón: define la forma (qué parámetros recibe, qué devuelve) y en tiempo de ejecución se conecta a un método real que cumpla esa forma.

La sintaxis clásica para declarar un delegado usa la palabra clave `delegate`:

```

using System;

class Programa
{
    // Declaración de un delegado: representa métodos que reciben int y devuelven
    int
    delegate int Operacion(int x);

    static int Duplicar(int n) { return n * 2; }
    static int ElevarAlCuadrado(int n) { return n * n; }

    static void Main()
    {
        // El delegado guarda una referencia al método Duplicar
        Operacion op = Duplicar;
        Console.WriteLine(op(5)); // Llama a Duplicar(5): muestra 10

        // Ahora el mismo delegado apunta a otro método compatible
        op = ElevarAlCuadrado;
        Console.WriteLine(op(5)); // Llama a ElevarAlCuadrado(5): muestra 25

        // También se puede invocar con Invoke (es lo mismo que op(5))
        Console.WriteLine(op.Invoke(5)); // Muestra 25
    }
}

```

¿Por qué querrías guardar un método en una variable? Porque te permite pasar comportamiento como parámetro. Imagina un método que procesa una lista de números, pero la operación exacta (duplicar, elevar al cuadrado, sumar diez) la decide quien llama al método. Sin delegados tendrías que escribir un método distinto para cada operación o usar un switch enorme. Con delegados, escribes el recorrido una sola vez y le pasas "qué hacer con cada elemento":

```

using System;

class Programa
{
    delegate int Operacion(int x);

    static int Duplicar(int n) { return n * 2; }
    static int SumarDiez(int n) { return n + 10; }

    // Este método NO sabe qué operación hará; la recibe como delegado
    static void ProcesarNumeros(int[] numeros, Operacion operacion)
    {
        foreach (int n in numeros)
        {
            // Llama al método que le pasaron, sea cual sea
            Console.WriteLine(operacion(n));
        }
    }

    static void Main()
    {
        int[] datos = { 1, 2, 3 };
        Console.WriteLine("Duplicando:");
        ProcesarNumeros(datos, Duplicar);
        Console.WriteLine("Sumando diez:");
        ProcesarNumeros(datos, SumarDiez);
    }
}

```

Este patrón (pasar comportamiento como argumento) es la base de LINQ, que ya conoces: cuando escribes `lista.Where(x => x > 5)`, le estás pasando un delegado (una lambda) al método `Where`, diciéndole "quédate con los elementos que cumplan esta condición". Sin delegados, LINQ no existiría.

Un delegado también puede apuntar a varios métodos a la vez: es un delegado multicast. Con `+=` agregas métodos a la lista de invocación y con `-=` los quitas. Al invocar el delegado, se llaman todos los métodos en orden:


```

using System;

class Programa
{
    delegate void Notificador(string mensaje);

    static void EnviarEmail(string m) { Console.WriteLine("[Email] " + m); }
    static void EnviarSMS(string m) { Console.WriteLine("[SMS] " + m); }
    static void GuardarLog(string m) { Console.WriteLine("[Log] " + m); }

    static void Main()
    {
        Notificador avisos = EnviarEmail;
        avisos += EnviarSMS;    // Agrega el segundo método
        avisos += GuardarLog;   // Agrega el tercero

        // Al invocar, se ejecutan los tres en orden
        avisos("El pedido está listo");
        // Muestra las tres líneas: [Email], [SMS] y [Log]

        avisos -= EnviarSMS;    // Quita el método de SMS
        avisos("Segundo aviso"); // Ahora solo Email y Log
    }
}

```

Los delegados multicast son el mecanismo que hay detrás de los eventos, como verás en un momento.

Func, Action y Predicate: delegados listos para usar

Declarar tu propio delegado con la palabra `delegate` funciona, pero en el código moderno casi nunca hace falta, porque .NET trae delegados genéricos que cubren todos los casos. Son tres y vale la pena memorizarlos.

`Func<T>` representa un método que DEVUELVE un valor. El último parámetro de tipo es siempre el tipo de retorno: `Func<int>` es un método sin parámetros que devuelve `int`; `Func<int, string>` es un método que recibe un `int` y devuelve un `string`; `Func<int, int, int>` recibe dos `int` y devuelve `int`. Puedes tener hasta 16 parámetros de entrada.

`Action<T>` representa un método que NO devuelve nada (`void`). `Action` es un método sin parámetros que no devuelve nada; `Action<string>` recibe un `string` y no devuelve nada; `Action<int, string>` recibe un `int` y un `string` y no devuelve nada.

Predicate<T> representa un método que recibe un T y devuelve bool (verdadero o falso). Se usa para condiciones y filtros: Predicate<int> es un método que recibe un int y dice si cumple o no una condición.

Con estos tres, el ejemplo anterior se escribe sin declarar ningún delegado:

```
using System;

class Programa
{
    static int Duplicar(int n) { return n * 2; }

    // Func<int, int>: recibe int, devuelve int (como el delegado Operacion)
    static void ProcesarNumeros(int[] numeros, Func<int, int> operacion)
    {
        foreach (int n in numeros)
        {
            Console.WriteLine(operacion(n));
        }
    }

    static void Main()
    {
        int[] datos = { 1, 2, 3, 4, 5, 6 };

        // Action<string>: recibe string, no devuelve nada
        Action<string> saludar = nombre => Console.WriteLine("Hola, " + nombre);
        saludar("Ana"); // Muestra: Hola, Ana

        // Predicate<int>: recibe int, devuelve bool
        Predicate<int> esPar = n => n % 2 == 0;
        Console.WriteLine(esPar(4)); // True
        Console.WriteLine(esPar(7)); // False

        // Func con el método Duplicar
        ProcesarNumeros(datos, Duplicar);
    }
}
```

Observa las lambdas que aparecieron: "nombre => Console.WriteLine(...)" y "n => n % 2 == 0". Esas flechitas => son expresiones lambda, y las veremos a fondo en la siguiente sección. Por ahora basta con saber que son una forma compacta de escribir un método pequeño justo donde lo necesitas, sin declararlo por separado.

La ventaja de Func, Action y Predicate es la interoperabilidad: todo el ecosistema .NET los usa. Los métodos de LINQ (Where, Select, OrderBy, ForEach en listas) esperan estos delegados, así que si los dominas, LINQ se vuelve natural. Cuando veas una firma como "IEnumerable<T> Where(Func<T, bool> predicado)", ya sabes leerla: "Where recibe una función que toma un T y devuelve bool, y filtra con ella".

Expresiones lambda: métodos de una línea con =>

Las expresiones lambda son la forma más concisa de crear un delegado. La sintaxis es: parámetros a la izquierda de =>, y el cuerpo a la derecha. Si el cuerpo es una sola expresión, su valor se devuelve automáticamente; si necesitas varias instrucciones, usas llaves y return explícito.

```
// Lambda de una expresión: recibe n, devuelve n * 2
Func<int, int> duplicar = n => n * 2;

// Lambda con varios parámetros (van entre paréntesis)
Func<int, int, int> sumar = (a, b) => a + b;

// Lambda con varias instrucciones (llaves + return)
Func<int, string> describir = n =>
{
    if (n % 2 == 0)
        return n + " es par";
    else
        return n + " es impar";
};

// Lambda sin parámetros (paréntesis vacíos)
Action aplaudir = () => Console.WriteLine("¡Bravo!");

// Lambda que no devuelve nada pero hace algo
Action<string> despedir = nombre =>
{
    Console.WriteLine("Adiós, " + nombre);
    Console.WriteLine("¡Nos vemos pronto!");
};
```

Las lambdas tienen un superpoder: capturan variables del contexto donde se crean (esto se llama closure o clausura). Mira:

```

using System;

class Programa
{
    static void Main()
    {
        int factor = 10; // Variable local del método Main

        // La lambda "captura" la variable factor aunque está fuera de ella
        Func<int, int> multiplicarPorFactor = n => n * factor;

        Console.WriteLine(multiplicarPorFactor(5)); // 50

        factor = 100; // Si cambia la variable, la lambda ve el cambio
        Console.WriteLine(multiplicarPorFactor(5)); // 500
    }
}

```

La lambda no copió el valor 10: capturó la variable misma. Esto es muy útil (por ejemplo, para configurar comportamiento con valores que solo se conocen en tiempo de ejecución), pero también es una fuente clásica de errores en ciclos: si creas lambdas dentro de un for que capturan la variable del ciclo, todas pueden terminar viendo el último valor. En C# moderno esto se corrigió para foreach, pero en for sigue siendo una trampa conocida. La lección: cuando una lambda captura una variable que cambia, asegúrate de entender qué valor verá cuando se ejecute, no cuando se creó.

Las lambdas son omnipresentes en C# moderno: LINQ, Task.Run, eventos, temporizadores, callbacks. Cada vez que veas =>, piensa "aquí estoy definiendo un pedacito de comportamiento que se ejecutará después o en otro contexto".

Eventos y el patrón publicador-suscriptor

Los eventos son delegados con reglas de etiqueta. Imagina una alarma contra incendios: la alarma (publicador) anuncia "¡hay humo!" y no le importa quién la escuche; las personas en el edificio (suscriptores) reaccionan cada una a su manera: unas salen corriendo, otras llaman a los bomberos. La alarma no mantiene una lista manual de personas ni les dice qué hacer: solo publica el aviso. Ese es el patrón publicador-suscriptor, y en C# se implementa con eventos.

La palabra clave event envuelve un delegado y le pone dos restricciones importantes: solo la clase que declara el evento puede dispararlo (invocarlo), y desde fuera solo se puede suscribir (+=) o desuscribir (-=). Esto protege el diseño: nadie externo puede "disparar" tu evento ni

borrar a todos tus suscriptores de un plumazo (con un delegado normal, alguien podría hacer "miDelegado = null" y dejar a todos sin aviso).

La convención de .NET para eventos usa el delegado EventHandler o EventHandler<TEventArgs>: el primer parámetro es el emisor (sender, de tipo object) y el segundo son los datos del evento (una clase que hereda de EventArgs). Veamos un ejemplo completo: un temporizador de cocina que avisa cuando la comida está lista.

```
using System;
using System.Threading;

// Datos del evento: qué información viaja con el aviso
class ComidaListaEventArgs : EventArgs
{
    public string Platillo { get; }
    public int MinutosCocinados { get; }

    public ComidaListaEventArgs(string platillo, int minutos)
    {
        Platillo = platillo;
        MinutosCocinados = minutos;
    }
}

// PUBLICADOR: la cocina que avisa cuando termina
class Cocina
{
    // Declaración del evento siguiendo la convención .NET
    public event EventHandler<ComidaListaEventArgs> ComidaLista;

    public void Cocinar(string platillo, int minutos)
    {
        Console.WriteLine("Cocinando " + platillo + "...");
        Thread.Sleep(minutos * 500); // Simula la cocción

        // Disparar el evento: avisar a todos los suscriptores
        // El ?. evita errores si nadie está suscrito
        ComidaLista?.Invoke(this, new ComidaListaEventArgs(platillo, minutos));
    }
}

class Programa
{
    static void Main()
    {
    }
```

```

Cocina cocina = new Cocina();

// SUSCRIPTOR 1: el mesero se suscribe al evento
cocina.ComidaLista += (sender, e) =>
{
    Console.WriteLine("Mesero: ¡" + e.Platillo + " listo! Voy por él.");
};

// SUSCRIPTOR 2: el cliente también se suscribe
cocina.ComidaLista += (sender, e) =>
{
    Console.WriteLine("Cliente: ¡Por fin! Mi " + e.Platillo +
        " después de " + e.MinutosCocinados + " minutos.");
};

cocina.Cocinar("pizza", 2);
// Al terminar, AMBOS suscriptores reaccionan automáticamente
}
}

```

Analiza las piezas. La clase `ComidaListaEventArgs` lleva los datos del evento (qué platillo, cuántos minutos). La clase `Cocina` declara el evento `ComidaLista` y lo dispara con `Invoke` cuando la comida está lista, pasando "this" como emisor y los datos. En `Main`, dos suscriptores se enganchan con `+=` usando lambdas. Cuando `Cocinar` termina, ambos métodos se ejecutan sin que la cocina sepa quiénes son. Si mañana agregas un tercer suscriptor (el gerente que anota la venta), la clase `Cocina` no cambia ni una línea: eso es bajo acoplamiento, una de las propiedades más valiosas del diseño de software.

El operador `?.` antes de `Invoke` es importante: si nadie se suscribió al evento, el delegado es `null` e invocarlo lanzaría `NullReferenceException`. "`ComidaLista?.Invoke(...)`" significa "si hay suscriptores, avísales; si no, no hagas nada". Es el patrón estándar y debes usarlo siempre al disparar eventos.

Los eventos también sirven para comunicar entre partes de un programa sin que se conozcan directamente. En una aplicación con interfaz gráfica, el botón publica el evento `Click` y tu código se suscribe; el botón no sabe nada de tu lógica de negocio y tu lógica no sabe nada del botón. En un juego, el jugador publica "`SaludBaja`" y la interfaz, el sonido y el sistema de logros reaccionan cada uno por su cuenta. Una vez que entiendes este patrón, empiezas a verlo en todas partes.

Un detalle de buenas prácticas: si tu evento no necesita datos adicionales, puedes usar el `EventHandler` simple (no genérico) con `EventArgs.Empty`. Y si defines tus propios delegados para eventos en código moderno, casi siempre es mejor usar `EventHandler<T>` directamente

en lugar de inventar una firma nueva: la convención (sender, e) la entiende cualquier programador de C#.

Genéricos: una sola clase para todos los tipos

Ya usaste genéricos desde las primeras unidades sin que te lo dijeran: cada vez que escribiste `List<string>` o `Dictionary<string, int>` estabas usando una clase genérica. Es momento de entender qué hay detrás de esos corchetes angulares `< >` y, más importante, de aprender a crear tus propios genéricos.

El problema que resuelven los genéricos es la duplicación. Imagina que necesitas una clase "Caja" que guarde un objeto y te lo devuelva después. Sin genéricos tendrías dos caminos malos. Camino uno: escribes `CajaDeEnteros`, `CajaDeCadenas`, `CajaDeProductos`... una clase por cada tipo, copiando y pegando el mismo código. Camino dos: escribes una sola clase `Caja` que guarda `object` (el tipo base de todo en C#). El camino dos parece inteligente hasta que lo usas: cada vez que sacas algo de la caja tienes que convertirlo (casting) al tipo real, lo cual es lento y peligroso, porque el compilador no puede verificar nada y puedes meter un texto en una caja que creías de números, con un error que explota al ejecutar el programa, no al compilarlo.

Los genéricos son el tercer camino, el bueno: escribes la clase una sola vez con un "marcador de tipo" llamado parámetro de tipo (por convención se llama `T`), y quien usa la clase decide qué tipo concreto reemplaza a `T`. El compilador genera, en efecto, una versión especializada para cada tipo, con verificación total en tiempo de compilación y sin conversiones costosas.

```

using System;

// Clase genérica: T es el parámetro de tipo, se decide al usarla
class Caja<T>
{
    private T contenido; // T se usa como cualquier tipo normal

    public void Guardar(T valor)
    {
        contenido = valor;
    }

    public T Obtener()
    {
        return contenido;
    }
}

class Programa
{
    static void Main()
    {
        // Caja de enteros: T se convierte en int
        Caja<int> cajaNumeros = new Caja<int>();
        cajaNumeros.Guardar(42);
        int numero = cajaNumeros.Obtener(); // Sin casting, ya es int
        Console.WriteLine(numero);

        // Caja de cadenas: T se convierte en string
        Caja<string> cajaTextos = new Caja<string>();
        cajaTextos.Guardar("Hola");
        string texto = cajaTextos.Obtener(); // Sin casting, ya es string

        // El compilador te protege: esto NO compila
        // cajaNumeros.Guardar("texto"); // Error: se esperaba int
    }
}

```

Esa última línea comentada es la magia: el error aparece al compilar, cuando es barato de corregir, no a las tres de la mañana en el servidor de producción. La seguridad de tipos en tiempo de compilación es el beneficio número uno de los genéricos.

Puedes usar varios parámetros de tipo si lo necesitas. El ejemplo clásico es un par clave-valor:


```

using System;

// Dos parámetros de tipo: TClave y TValor
class Pareja<TClave, TValor>
{
    public TClave Clave { get; set; }
    public TValor Valor { get; set; }

    public Pareja(TClave clave, TValor valor)
    {
        Clave = clave;
        Valor = valor;
    }

    public void Mostrar()
    {
        Console.WriteLine(Clave + " -> " + Valor);
    }
}

class Programa
{
    static void Main()
    {
        Pareja<string, int> edad = new Pareja<string, int>("Ana", 30);
        edad.Mostrar(); // Ana -> 30

        Pareja<int, string> codigo = new Pareja<int, string>(404, "No encontrado");
        codigo.Mostrar(); // 404 -> No encontrado
    }
}

```

Dictionary<TClave, TValor>, que ya conoces, es exactamente esto llevado a su máxima expresión: una colección genérica con dos parámetros de tipo.

Métodos genéricos

No solo las clases pueden ser genéricas: los métodos también. Un método genérico define sus propios parámetros de tipo, y el compilador normalmente los infiere de los argumentos, así que ni siquiera tienes que escribirlos. El ejemplo más famoso es un método que intercambia dos valores, que funciona con cualquier tipo:

```

using System;

class Utilidades
{
    // Método genérico: funciona con cualquier tipo T
    public static void Intercambiar<T>(ref T a, ref T b)
    {
        T temporal = a;
        a = b;
        b = temporal;
    }

    // Otro ejemplo: imprime cualquier arreglo, del tipo que sea
    public static void ImprimirArreglo<T>(T[] arreglo)
    {
        foreach (T elemento in arreglo)
        {
            Console.Write(elemento + " ");
        }
        Console.WriteLine();
    }
}

class Programa
{
    static void Main()
    {
        int x = 1, y = 2;
        // El compilador infiere T = int por los argumentos
        Utilidades.Intercambiar(ref x, ref y);
        Console.WriteLine(x + ", " + y); // 2, 1

        string a = "hola", b = "mundo";
        // Aquí infiere T = string
        Utilidades.Intercambiar(ref a, ref b);
        Console.WriteLine(a + ", " + b); // mundo, hola

        int[] numeros = { 1, 2, 3 };
        string[] nombres = { "Ana", "Luis" };
        Utilidades.ImprimirArreglo(numeros); // 1 2 3
        Utilidades.ImprimirArreglo(nombres); // Ana Luis
    }
}

```

Observa que al llamar a Intercambiar no escribimos Intercambiar<int>: el compilador deduce T a partir de los argumentos (inferencia de tipos). Si alguna vez la inferencia no puede deducirlo, puedes especificarlo explícitamente: Utilidades.Intercambiar<int>(ref x, ref y). Los métodos genéricos son ideales para utilidades transversales: conversiones, comparaciones, búsquedas, clonados.

Restricciones con where T: ponerle reglas al tipo

A veces tu clase genérica necesita hacer algo específico con T: crear instancias nuevas, comparar valores, o llamar a un método que solo existe en ciertos tipos. Como T puede ser cualquier cosa, el compilador no te deja hacer esas operaciones... a menos que le prometas, con una restricción where T, que T cumplirá ciertos requisitos. Las restricciones más usadas son:

- where T : class - T debe ser un tipo por referencia (una clase).
- where T : struct - T debe ser un tipo por valor (int, double, DateTime, etc.).
- where T : new() - T debe tener un constructor sin parámetros, para poder hacer new T().
- where T : NombreBase - T debe heredar de la clase NombreBase.
- where T : IAlgunaInterfaz - T debe implementar esa interfaz.
- where T : notnull - T no puede ser un tipo anulable (novedad de C# 8).

El caso más didáctico es new(): sin la restricción, no puedes escribir "new T()" dentro de tu clase genérica, porque ¿y si T es un tipo sin constructor vacío? Con la restricción, el compilador lo permite porque le garantizaste que T siempre tendrá ese constructor:

```

using System;

// T debe tener constructor sin parámetros (new()) y ser clase (class)
class Fabrica<T> where T : class, new()
{
    public T Crear()
    {
        // Gracias a la restricción, esto compila sin problemas
        T objeto = new T();
        Console.WriteLine("Creado un " + typeof(T).Name);
        return objeto;
    }
}

class Producto
{
    public string Nombre { get; set; } = "Sin nombre";
}

class Programa
{
    static void Main()
    {
        Fabrica<Producto> fabrica = new Fabrica<Producto>();
        Producto p = fabrica.Crear(); // Creado un Producto
        Console.WriteLine(p.Nombre); // Sin nombre
    }
}

```

Otro uso típico es restringir a una interfaz para poder llamar sus métodos con seguridad:

```

using System;
using System.Collections.Generic;

interface INombrable
{
    string Nombre { get; }
}

class Estudiante : INombrable
{
    public string Nombre { get; set; }
}

class Curso : INombrable
{
    public string Nombre { get; set; }
}

class Programa
{
    // T debe implementar INombrable, así que podemos usar .Nombre sin miedo
    static void MostrarNombres<T>(List<T> lista) where T : INombrable
    {
        foreach (T elemento in lista)
        {
            Console.WriteLine("- " + elemento.Nombre);
        }
    }

    static void Main()
    {
        List<Estudiante> estudiantes = new List<Estudiante>
        {
            new Estudiante { Nombre = "Ana" },
            new Estudiante { Nombre = "Luis" }
        };
        MostrarNombres(estudiantes); // Funciona porque Estudiante es INombrable

        // Esto NO compilaría: int no implementa INombrable
        // MostrarNombres(new List<int> { 1, 2, 3 });
    }
}

```

Las restricciones son un contrato: le dices al compilador "solo acepto tipos que cumplan esto" y a cambio él te deja usar las capacidades prometidas dentro de tu código genérico. Úsalas cuando tu código genérico necesite hacer algo más que guardar y devolver valores.

Por qué List<T> es mejor que ArrayList

Para apreciar los genéricos, vale la pena mirar al pasado. En las primeras versiones de .NET no había genéricos, y la colección de propósito general era ArrayList, que guarda elementos de tipo object. Como todo en C# hereda de object, podías meter cualquier cosa: enteros, cadenas, fechas, todo revuelto. El precio era triple.

Primero, el boxing y unboxing: cada vez que metías un int (tipo por valor) en un ArrayList, .NET tenía que "empaquetarlo" (boxing) en un objeto del montón (heap), y cada vez que lo sacabas, "desempaquetarlo" (unboxing) con una conversión explícita. Esas operaciones cuestan tiempo y generan basura para el recolector de memoria. Con List<int>, el entero se guarda directamente, sin empaquetado.

Segundo, la falta de seguridad de tipos: nada impedía meter un string en un ArrayList que tú considerabas "de enteros". El error aparecía al ejecutar, cuando intentabas convertirlo, con una InvalidCastException en el peor momento posible.

Tercero, el código sucio: cada lectura requería un cast explícito (int)lista[0], lo que llenaba el programa de conversiones y lo hacía difícil de leer.

```

using System;
using System.Collections;

class Programa
{
    static void Main()
    {
        // FORMA ANTIGUA: ArrayList (no usar en código nuevo)
        ArrayList vieja = new ArrayList();
        vieja.Add(10);
        vieja.Add("texto infiltrado"); // El compilador no se queja...
        // int n = (int)vieja[1]; // ...pero esto explota al ejecutar

        // FORMA MODERNA: List<T> con genéricos
        List<int> moderna = new List<int>();
        moderna.Add(10);
        // moderna.Add("texto"); // ERROR de compilación: imposible equivocarse
        int m = moderna[0]; // Sin cast, directo y rápido
        Console.WriteLine(m);
    }
}

```

La lección es general: siempre que veas una clase antigua no genérica (ArrayList, Hashtable, Queue sin tipo), prefiere su equivalente genérico (List<T>, Dictionary<TKey, TValue>, Queue<T>). Son más rápidas, más seguras y el código queda más limpio. Los genéricos no son un lujo académico: son la razón por la que las colecciones modernas de .NET son tan buenas.

Errores comunes y buenas prácticas con genéricos

El error más común es intentar usar operadores o miembros específicos sobre T sin restricción. Por ejemplo, dentro de un método genérico no puedes escribir "return a + b" si T es genérico, porque el compilador no sabe si T soporta el operador +. La solución es restringir T a algo que garantice la operación, o usar interfaces como IComparable<T> para comparaciones.

Otro error es abusar de los genéricos donde no aportan nada: si tu clase solo va a usarse con string, no la hagas genérica "por si acaso". Los genéricos brillan cuando el mismo código debe servir para muchos tipos; si solo hay un tipo, la generalidad añade complejidad sin beneficio.

En cuanto a nomenclatura, usa T para un solo parámetro de tipo; con varios, usa nombres descriptivos como TClave, TValor, TEntrada, TSalida, o al menos T, U, V en ese orden. Evita nombres de una letra que no sean T cuando el código lo leerá otra persona.

Finalmente, recuerda que los genéricos y la reflexión a veces chocan: obtener el tipo real con `typeof(T)` o `elemento.GetType()` funciona bien, pero crear instancias requiere la restricción `new()`, como ya viste. Y un detalle fino: cada combinación de tipos genera una versión distinta en tiempo de ejecución (`List<int>` y `List<string>` son tipos diferentes y no se pueden convertir entre sí), lo cual es precisamente lo que garantiza la seguridad de tipos.

Con esto cierras la explicación detallada de los tres grandes temas. En la siguiente sección los verás en acción, combinados en ejemplos paso a paso que integran `async`, delegados y genéricos como lo haría un programa real.

Ejemplos paso a paso

Ejemplo 1: Descargador asíncrono con progreso (async + eventos)

Vamos a construir un simulador de descargas que combina `async/await` con eventos: la clase `Descargador` hace el trabajo pesado de forma asíncrona y publica eventos de progreso a los que cualquiera puede suscribirse. Es el patrón que usan las aplicaciones reales para barras de progreso.

```
using System;
using System.Threading.Tasks;

// Datos del evento de progreso: porcentaje y archivo
class ProgresoEventArgs : EventArgs
{
    public string Archivo { get; }
    public int Porcentaje { get; }

    public ProgresoEventArgs(string archivo, int porcentaje)
    {
        Archivo = archivo;
        Porcentaje = porcentaje;
    }
}

class Descargador
{
    // Evento: avisa cada vez que avanza la descarga
    public event EventHandler<ProgresoEventArgs> ProgresoCambio;

    // Método async: simula descargar un archivo por partes
    public async Task DescargarAsync(string archivo)
```



```

    {
        // Simulamos 10 partes de la descarga
        for (int parte = 1; parte <= 10; parte++)
        {
            // Espera asíncrona de 300 ms: simula la red SIN bloquear
            await Task.Delay(300);

            // Calcula el porcentaje y dispara el evento
            int porcentaje = parte * 10;
            ProgresoCambio?.Invoke(this, new ProgresoEventArgs(archivo, porcentaje));
        }
    }
}

class Programa
{
    static async Task Main(string[] args)
    {
        Descargador d = new Descargador();

        // Nos suscribimos: cada avance dibuja la barra de progreso
        d.ProgresoCambio += (sender, e) =>
        {
            Console.WriteLine(e.Archivo + ": " + e.Porcentaje + "% completado");
        };

        Console.WriteLine("Iniciando descarga...");
        // await: esperamos sin bloquear; los eventos llegan solos
        await d.DescargarAsync("video.mp4");
        Console.WriteLine("¡Descarga terminada!");
    }
}

```

Línea por línea, lo importante: `DescargarAsync` es `async` y devuelve `Task`; dentro del ciclo, `await Task.Delay(300)` pausa sin bloquear; después de cada pausa dispara el evento `ProgresoCambio` con `?.Invoke`; en `Main` nos suscribimos con una lambda que imprime el progreso; el `await` final espera a que todo termine. Si quisieras descargar tres archivos a la vez, lanzarías tres tareas y usarías `Task.WhenAll`: cada una reportaría su progreso por eventos de forma independiente.

Ejemplo 2: Filtros con Func y lambdas (delegados en acción)

Ahora un ejemplo que muestra el poder de pasar comportamiento: un buscador de productos que recibe el criterio de filtrado como `Func<Producto, bool>`. El mismo método sirve para buscar por precio, por categoría o por cualquier condición que se te ocurra, sin modificarlo.

```
using System;
using System.Collections.Generic;

class Producto
{
    public string Nombre { get; set; }
    public string Categoria { get; set; }
    public double Precio { get; set; }

    public Producto(string nombre, string categoria, double precio)
    {
        Nombre = nombre;
        Categoria = categoria;
        Precio = precio;
    }
}

class Tienda
{
    private List<Producto> productos = new List<Producto>();

    public void Agregar(Producto p)
    {
        productos.Add(p);
    }

    // Método genérico en el criterio: recibe QUÉ filtrar como delegado
    public List<Producto> Buscar(Func<Producto, bool> criterio)
    {
        List<Producto> resultado = new List<Producto>();
        foreach (Producto p in productos)
        {
            // El delegado decide si el producto pasa el filtro
            if (criterio(p))
            {
                resultado.Add(p);
            }
        }
        return resultado;
    }
}
```

```

    }
}

class Programa
{
    static void Mostrar(List<Producto> lista)
    {
        foreach (Producto p in lista)
        {
            Console.WriteLine("- " + p.Nombre + " (" + p.Categoria + "): $" +
p.Precio);
        }
    }

    static void Main()
    {
        Tienda tienda = new Tienda();
        tienda.Agregar(new Producto("Laptop", "Electrónica", 15000));
        tienda.Agregar(new Producto("Mouse", "Electrónica", 300));
        tienda.Agregar(new Producto("Silla", "Muebles", 2500));
        tienda.Agregar(new Producto("Teclado", "Electrónica", 800));

        Console.WriteLine("Baratos (menos de $1000):");
        // Pasamos el criterio como lambda: p => p.Precio < 1000
        Mostrar(tienda.Buscar(p => p.Precio < 1000));

        Console.WriteLine("Electrónica:");
        Mostrar(tienda.Buscar(p => p.Categoria == "Electrónica"));

        Console.WriteLine("Nombre largo (más de 6 letras):");
        Mostrar(tienda.Buscar(p => p.Nombre.Length > 6));
    }
}

```

El método `Buscar` no sabe nada de precios ni categorías: solo aplica el criterio que le pasan. Cada llamada en `Main` le entrega una lambda distinta. Si mañana necesitas "productos caros en oferta", no tocas la clase `Tienda`: escribes otra lambda. Esto es exactamente cómo funciona LINQ por dentro (Where recibe un `Func<T, bool>`), así que ya entiendes la maquinaria.

Ejemplo 3: Pila genérica con restricciones (genéricos en acción)

Cerremos los ejemplos con una estructura de datos genérica completa: una `Pila<T>` (stack: último en entrar, primero en salir) con restricción `new()` para poder crear elementos por defecto, y un método genérico de utilidad. Incluye manejo de errores para pila vacía.

```

using System;
using System.Collections.Generic;

// Pila genérica: T debe tener constructor sin parámetros
class Pila<T> where T : new()
{
    private List<T> elementos = new List<T>();

    // Apilar: agrega al final
    public void Apilar(T elemento)
    {
        elementos.Add(elemento);
    }

    // Desapilar: quita y devuelve el último; error si está vacía
    public T Desapilar()
    {
        if (elementos.Count == 0)
        {
            throw new InvalidOperationException("La pila está vacía.");
        }
        int ultimo = elementos.Count - 1;
        T valor = elementos[ultimo];
        elementos.RemoveAt(ultimo);
        return valor;
    }

    // Ver el tope sin quitarlo
    public T VerTope()
    {
        if (elementos.Count == 0)
        {
            throw new InvalidOperationException("La pila está vacía.");
        }
        return elementos[elementos.Count - 1];
    }

    public int Cantidad
    {
        get { return elementos.Count; }
    }

    // Gracias a where T : new() podemos crear un T vacío
    public T CrearVacio()
    {

```

```

        return new T();
    }
}

class Punto
{
    public int X { get; set; }
    public int Y { get; set; }
    // Constructor sin parámetros: cumple la restricción new()
    public Punto() { X = 0; Y = 0; }
}

class Programa
{
    static void Main()
    {
        // Pila de enteros
        Pila<int> pilaNumeros = new Pila<int>();
        pilaNumeros.Apilar(10);
        pilaNumeros.Apilar(20);
        pilaNumeros.Apilar(30);
        Console.WriteLine("Tope: " + pilaNumeros.VerTope()); // 30
        Console.WriteLine("Desapilado: " + pilaNumeros.Desapilar()); // 30
        Console.WriteLine("Quedan: " + pilaNumeros.Cantidad); // 2

        // Pila de puntos (tipo personalizado)
        Pila<Punto> pilaPuntos = new Pila<Punto>();
        pilaPuntos.Apilar(new Punto { X = 5, Y = 7 });
        Punto vacio = pilaPuntos.CrearVacio(); // new T() en acción
        Console.WriteLine("Punto vacío: (" + vacio.X + ", " + vacio.Y + ")");

        // Manejo del error de pila vacía
        Pila<string> pilaTextos = new Pila<string>();
        try
        {
            pilaTextos.Desapilar();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error controlado: " + ex.Message);
        }
    }
}

```

Una sola clase Pila<T> sirve para enteros, puntos, cadenas y cualquier tipo con constructor vacío. Sin genéricos habrías escrito (o copiado) esta clase tres veces, o usado object con casts por todas partes. Esta es la reutilización real que los genéricos hacen posible.

Ejemplo 4: Todo junto - procesador asíncrono con callbacks genéricos

Para cerrar, un ejemplo que integra los tres temas: un procesador que ejecuta una lista de tareas asíncronas, permite registrar callbacks (delegados) que se ejecutan al terminar cada una, y usa genéricos para que el resultado pueda ser de cualquier tipo.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

class Procesador<T>
{
    // Callback genérico: recibe el resultado de tipo T cuando termina
    public event Action<T> TareaCompletada;

    // Ejecuta varias operaciones asíncronas y avisa por cada una
    public async Task ProcesarTodas(List<Func<Task<T>>> operaciones)
    {
        List<Task<T>> tareasEnCurso = new List<Task<T>>();

        // Lanzamos todas las operaciones sin esperar (arrancan en paralelo)
        foreach (var operacion in operaciones)
        {
            tareasEnCurso.Add(operacion());
        }

        // Esperamos a que terminen todas juntas
        T[] resultados = await Task.WhenAll(tareasEnCurso);

        // Avisamos a los suscriptores por cada resultado
        foreach (T resultado in resultados)
        {
            TareaCompletada?.Invoke(resultado);
        }
    }
}

class Programa
{
    // Operación asíncrona que devuelve int
```

```

static async Task<int> CalcularCuadrado(int n)
{
    await Task.Delay(500); // Simula trabajo
    return n * n;
}

static async Task Main(string[] args)
{
    Procesador<int> procesador = new Procesador<int>();

    // Suscribimos un callback: qué hacer con cada resultado
    procesador.TareaCompletada += resultado =>
    {
        Console.WriteLine("Recibido: " + resultado);
    };

    // Lista de operaciones (delegados que devuelven Task<int>)
    List<Func<Task<int>>> operaciones = new List<Func<Task<int>>>
    {
        () => CalcularCuadrado(3),
        () => CalcularCuadrado(5),
        () => CalcularCuadrado(7)
    };

    Console.WriteLine("Procesando en paralelo...");
    await procesador.ProcesarTodas(operaciones);
    Console.WriteLine("Todo terminado.");
}
}

```

Fíjate cómo conviven: `Procesador<T>` es genérico (funciona con `int`, `string` o lo que sea), recibe `Func<Task<T>>` (delegados que representan operaciones asíncronas), las lanza todas, las espera con `Task.WhenAll` y notifica con un evento `Action<T>`. Este es el nivel de composición que encontrarás en bibliotecas profesionales: genéricos para la reutilización, delegados para el comportamiento configurable y `async` para no bloquear.

Práctica

Es momento de ensuciarte las manos. Estos ejercicios están ordenados de menor a mayor dificultad y cubren los tres temas de la unidad. Intenta resolver cada uno por tu cuenta antes de mirar la solución: la asincronía y los delegados solo se dominan escribiéndolos, no leyéndolos.

1. Cronómetro paralelo. Escribe un programa con un método async llamado `EsperarYSaludarAsync(string nombre, int segundos)` que espere los segundos indicados con `Task.Delay` y luego imprima "Hola, {nombre}". En Main, lanza tres llamadas al mismo tiempo (Ana 1 segundo, Luis 2 segundos, María 3 segundos) usando `Task.WhenAll`, y mide con un cronómetro (puedes usar `DateTime.Now` antes y después) cuánto tarda todo. Comprueba que el total sea ~3 segundos y no ~6.
2. Reintento con espera. Escribe un método async `OperacionFragilAsync` que simule una operación que falla las primeras dos veces que se llama y tiene éxito a la tercera (usa un contador static y lanza una `Exception` cuando el contador sea menor que 3). Luego escribe otro método async `EjecutarConReintentosAsync` que intente llamar a la operación hasta 4 veces: si falla, que espere 1 segundo con `Task.Delay`, muestre "Reintentando...", y vuelva a intentar; si tiene éxito, que muestre el resultado. Este patrón de reintento con pausa es muy usado en llamadas a APIs reales.
3. Alarma de temperatura. Crea una clase `SensorTemperatura` con un evento `TemperaturaAlta` del tipo `EventHandler<double>` (puedes usar `EventHandler<T>` con `T = double` directamente, o crear tu `EventArgs`; ambas valen). La clase debe tener un método async `MonitorearAsync` que genere 10 lecturas aleatorias de temperatura entre 15 y 40 grados (usa `Random`), esperando 500 ms entre cada una con `Task.Delay`. Cada vez que una lectura supere los 30 grados, debe disparar el evento pasando la temperatura. En Main, suscribe dos manejadores: uno que imprima "¡ALARMA! Temperatura alta: X grados" y otro que cuente cuántas alarmas hubo y al final imprima el total.
4. Calculadora de estadísticas con delegados. Escribe una clase `Estadisticas` con un método `double Calcular(List<double> datos, Func<List<double>, double> operacion)` que simplemente aplique el delegado a la lista y devuelva el resultado. En Main, crea una lista de calificaciones y llama a `Calcular` tres veces con lambdas distintas: una que calcule el promedio, una que encuentre el valor máximo y una que cuente cuántos valores son mayores o iguales a 6 (aprobados). No modifiques la clase `Estadisticas`: toda la lógica debe ir en las lambdas.
5. Cola genérica. Implementa una clase `Cola<T>` (queue: primero en entrar, primero en salir) con los métodos `Encolar(T elemento)`, `Desencolar()` (quita y devuelve el primero, lanza `InvalidOperationException` si está vacía), `VerPrimero()` y una propiedad `Cantidad`. Usa una `List<T>` por dentro. Pruébala con `Cola<int>` y con `Cola<string>` en Main, incluyendo el manejo del error al desencolar de una cola vacía.
6. Máximo genérico con restricción. Escribe un método genérico `T EncontrarMaximo<T>(List<T> lista)` que devuelva el elemento más grande de la lista. Como necesita comparar elementos, agrega la restricción `where T : IComparable<T>` y

usa el método `CompareTo` para las comparaciones. Si la lista está vacía, lanza un `ArgumentException`. Pruébalo con una lista de `int`, una de `string` (ordena alfabéticamente) y una de `double`.

7. Reto integrador: noticiero asíncrono. Combina todo: crea una clase `Noticiero<T>` genérica con un evento `Action<T> NoticiaPublicada`. Agrega un método `async PublicarVariasAsync(List<Func<Task<T>>> generadores)` que lance todos los generadores en paralelo, espere con `Task.WhenAll` y dispare el evento por cada resultado. En `Main`, crea tres generadores `async` que simulen "obtener" noticias (cada uno espera un tiempo distinto con `Task.Delay` y devuelve un `string` con la noticia), suscribe al evento un manejador que imprima cada noticia con la hora actual, y ejecuta todo. Este ejercicio es un mini-modelo de cómo funcionan los sistemas reales de mensajería.

Soluciones

Solución 1: Cronómetro paralelo. La clave es iniciar las tres tareas sin `await` y luego esperarlas juntas con `WhenAll`. Si hicieras `await` una por una, tardaría 6 segundos.

```

using System;
using System.Threading.Tasks;

class Programa
{
    // Método async con sufijo Async, como manda la convención
    static async Task EsperarYSaludarAsync(string nombre, int segundos)
    {
        // Task.Delay espera sin bloquear el hilo
        await Task.Delay(segundos * 1000);
        Console.WriteLine("Hola, " + nombre);
    }

    static async Task Main(string[] args)
    {
        DateTime inicio = DateTime.Now;

        // Las tres tareas ARRANCAN aquí, sin await todavía: corren "a la vez"
        Task t1 = EsperarYSaludarAsync("Ana", 1);
        Task t2 = EsperarYSaludarAsync("Luis", 2);
        Task t3 = EsperarYSaludarAsync("María", 3);

        // Un solo await las espera a TODAS juntas
        await Task.WhenAll(t1, t2, t3);

        double segundosTotales = (DateTime.Now - inicio).TotalSeconds;
        Console.WriteLine("Tiempo total: " + segundosTotales.ToString("0.0") + " segundos");
        // Verás ~3.0 segundos, no 6: el await no bloquea, las esperas se traslapan
    }
}

```

Explicación: cada llamada a `EsperarYSaludarAsync` devuelve un `Task` inmediatamente y la espera de cada una ocurre de forma independiente. `Task.WhenAll` crea un `Task` que termina cuando las tres terminan, así que el tiempo total es el de la más lenta (3 segundos) en lugar de la suma. Este es el patrón estándar para paralelizar operaciones de espera.

Solución 2: Reintento con espera. El ciclo `for` controla los intentos; el `try/catch` dentro del ciclo atrapa el fallo, espera con `Task.Delay` y continúa; si tiene éxito, el `return` sale del método.

```

using System;
using System.Threading.Tasks;

class Programa

```

```

{
    static int intentosOperacion = 0;

    // Simula una operación que falla 2 veces y funciona a la tercera
    static async Task<string> OperacionFragilAsync()
    {
        intentosOperacion++;
        await Task.Delay(500); // Simula trabajo real
        if (intentosOperacion < 3)
        {
            throw new Exception("Fallo simulado número " + intentosOperacion);
        }
        return "¡Éxito en el intento " + intentosOperacion + "!";
    }

    static async Task EjecutarConReintentosAsync()
    {
        int maxIntentos = 4;
        for (int intento = 1; intento <= maxIntentos; intento++)
        {
            try
            {
                // Si funciona, mostramos el resultado y salimos
                string resultado = await OperacionFragilAsync();
                Console.WriteLine(resultado);
                return;
            }
            catch (Exception ex)
            {
                Console.WriteLine("Intento " + intento + " falló: " + ex.Message);
                if (intento < maxIntentos)
                {
                    Console.WriteLine("Reintentando en 1 segundo...");
                    // Pausa asíncrona antes del siguiente intento
                    await Task.Delay(1000);
                }
            }
        }
        Console.WriteLine("Se agotaron los intentos.");
    }

    static async Task Main(string[] args)
    {
        await EjecutarConReintentosAsync();
    }
}

```

```
}
```

Explicación: el patrón es "intenta, si falla espera y reintenta". El `await Task.Delay(1000)` es crucial: con `Thread.Sleep` bloquearías el hilo durante la espera. En APIs reales se suele agregar espera exponencial (1s, 2s, 4s...) y un `CancellationToken` para cancelar, pero la estructura base es esta.

Solución 3: Alarma de temperatura. Combina eventos con async: el sensor publica y los suscriptores reaccionan. Usamos `EventHandler<double>` directamente para simplificar.

```
using System;
using System.Threading.Tasks;

class SensorTemperatura
{
    // Evento genérico: el dato del evento es directamente un double
    public event EventHandler<double> TemperaturaAlta;

    private Random random = new Random();

    public async Task MonitorearAsync()
    {
        for (int i = 1; i <= 10; i++)
        {
            // Temperatura aleatoria entre 15.0 y 40.0
            double temp = 15 + random.NextDouble() * 25;
            Console.WriteLine("Lectura " + i + ": " + temp.ToString("0.0") + "°C");

            if (temp > 30)
            {
                // Dispara el evento solo si hay suscriptores (?)
                TemperaturaAlta?.Invoke(this, temp);
            }

            await Task.Delay(500); // Espera asíncrona entre lecturas
        }
    }
}

class Programa
{
    static async Task Main(string[] args)
    {
        SensorTemperatura sensor = new SensorTemperatura();
        int totalAlarmas = 0;
    }
}
```

```

        // Suscriptor 1: muestra la alarma (lambda con dos parámetros)
        sensor.TemperaturaAlta += (sender, temp) =>
        {
            Console.WriteLine(" ¡ALARMA! Temperatura alta: " + temp.ToString("0.0") +
"°C");
        };

        // Suscriptor 2: cuenta las alarmas (captura la variable totalAlarmas)
        sensor.TemperaturaAlta += (sender, temp) =>
        {
            totalAlarmas++;
        };

        await sensor.MonitorearAsync();
        Console.WriteLine("Monitoreo terminado. Total de alarmas: " + totalAlarmas);
    }
}

```

Explicación: el sensor (publicador) no sabe quién lo escucha; solo dispara TemperaturaAlta cuando corresponde. Los dos suscriptores reaccionan de forma independiente: uno avisa y otro cuenta. La lambda del contador captura la variable totalAlarmas del Main (closure). Si mañana agregas un suscriptor que guarde en archivo, la clase SensorTemperatura no cambia.

Solución 4: Calculadora de estadísticas con delegados. La clase Estadisticas es mínima; todo el comportamiento viaja en las lambdas.

```

using System;
using System.Collections.Generic;
using System.Linq;

class Estadisticas
{
    // Recibe los datos y QUÉ calcular como delegado Func
    public double Calcular(List<double> datos, Func<List<double>, double>
operacion)
    {
        return operacion(datos);
    }
}

class Programa
{
    static void Main()
    {
        Estadisticas est = new Estadisticas();
        List<double> calificaciones = new List<double> { 8.5, 6.0, 9.2, 4.5, 7.8,
5.9 };

        // Lambda 1: promedio (usamos LINQ por comodidad)
        double promedio = est.Calcular(calificaciones, lista => lista.Average());
        Console.WriteLine("Promedio: " + promedio.ToString("0.00"));

        // Lambda 2: valor máximo
        double maximo = est.Calcular(calificaciones, lista => lista.Max());
        Console.WriteLine("Máximo: " + maximo);

        // Lambda 3: cuántos aprobaron (>= 6). Count devuelve int, lo convertimos
        double aprobados = est.Calcular(calificaciones,
            lista => lista.Count(c => c >= 6));
        Console.WriteLine("Aprobados: " + aprobados);
    }
}

```

Explicación: Calcular no sabe nada de promedios ni máximos; solo aplica el delegado que recibe. Cada llamada le pasa una lambda distinta con la lógica. Nota que `Func<List<double>, double>` describe "método que recibe lista y devuelve double", y `Count` devuelve `int` pero se convierte implícitamente a `double` al devolverlo (conversión implícita de `int` a `double`). Este es el mismo principio de LINQ: `Where`, `Select` y `Average` reciben tu lógica como delegados.

Solución 5: Cola genérica. Una sola clase sirve para cualquier tipo; la List<T> interna hace el trabajo pesado.

```
using System;
using System.Collections.Generic;

class Cola<T>
{
    // Almacenamiento interno genérico
    private List<T> elementos = new List<T>();

    // Encolar: agrega al final (los nuevos van atrás)
    public void Encolar(T elemento)
    {
        elementos.Add(elemento);
    }

    // Desencolar: quita y devuelve el PRIMERO (los viejos salen antes)
    public T Desencolar()
    {
        if (elementos.Count == 0)
        {
            throw new InvalidOperationException("La cola está vacía.");
        }
        T primero = elementos[0];
        elementos.RemoveAt(0);
        return primero;
    }

    // Ver el primero sin quitarlo
    public T VerPrimero()
    {
        if (elementos.Count == 0)
        {
            throw new InvalidOperationException("La cola está vacía.");
        }
        return elementos[0];
    }

    public int Cantidad
    {
        get { return elementos.Count; }
    }
}
```

```

class Programa
{
    static void Main()
    {
        // Cola de enteros: simula turnos
        Cola<int> turnos = new Cola<int>();
        turnos.Encolar(101);
        turnos.Encolar(102);
        turnos.Encolar(103);
        Console.WriteLine("Atendiendo turno: " + turnos.Desencolar()); // 101
        Console.WriteLine("Siguiente: " + turnos.VerPrimero()); // 102
        Console.WriteLine("En espera: " + turnos.Cantidad); // 2

        // Cola de strings: misma clase, otro tipo
        Cola<string> fila = new Cola<string>();
        fila.Encolar("Ana");
        fila.Encolar("Luis");
        Console.WriteLine("Pasa: " + fila.Desencolar()); // Ana

        // Error controlado al desencolar de cola vacía
        Cola<int> vacia = new Cola<int>();
        try
        {
            vacia.Desencolar();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error controlado: " + ex.Message);
        }
    }
}

```

Explicación: la diferencia con la Pila<T> del ejemplo es el orden de salida: la pila saca el último (LIFO) y la cola saca el primero (FIFO). El resto de la maquinaria genérica es idéntica: una sola clase, seguridad de tipos en compilación, sin casts. En .NET real existe Queue<T> que hace esto de forma optimizada; este ejercicio te enseña cómo está construida por dentro.

Solución 6: Máximo genérico con restricción. La restricción where T : IComparable<T> es lo que permite comparar elementos de un tipo desconocido.


```

using System;
using System.Collections.Generic;

class Programa
{
    // T debe ser comparable consigo mismo: así podemos usar CompareTo
    static T EncontrarMaximo<T>(List<T> lista) where T : IComparable<T>
    {
        if (lista.Count == 0)
        {
            throw new ArgumentException("La lista no puede estar vacía.");
        }

        T maximo = lista[0];
        foreach (T elemento in lista)
        {
            // CompareTo devuelve > 0 si elemento es mayor que maximo
            if (elemento.CompareTo(maximo) > 0)
            {
                maximo = elemento;
            }
        }
        return maximo;
    }

    static void Main()
    {
        List<int> numeros = new List<int> { 4, 17, 2, 9, 13 };
        Console.WriteLine("Máximo int: " + EncontrarMaximo(numeros)); // 17

        List<string> nombres = new List<string> { "Ana", "Pedro", "Lucía" };
        Console.WriteLine("Máximo string: " + EncontrarMaximo(nombres)); // Pedro

        List<double> precios = new List<double> { 19.99, 5.50, 42.10 };
        Console.WriteLine("Máximo double: " + EncontrarMaximo(precios)); // 42.1

        // Sin la restricción, elemento.CompareTo no compilaría:
        // el compilador no sabría que T se puede comparar.
        // int, string y double implementan IComparable<T>, por eso funciona.
    }
}

```

Explicación: `IComparable<T>` es la interfaz que dice "sé compararme con otro T" mediante `CompareTo`. Al exigirla con `where T`, el método puede ordenar cualquier tipo comparable:

números, cadenas (orden alfabético), fechas, y tus propias clases si implementan la interfaz. Sin la restricción, el compilador rechazaría CompareTo porque T podría ser cualquier cosa. Es el ejemplo perfecto de para qué sirven las restricciones: te dan capacidades a cambio de un contrato.

Solución 7: Reto integrador, noticiero asíncrono. Une genéricos (Noticiero<T>), delegados (Func<Task<T>> y Action<T>) y async (WhenAll).

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

// Noticiero genérico: puede publicar noticias de cualquier tipo T
class Noticiero<T>
{
    // Evento con Action<T>: avisa con la noticia de tipo T
    public event Action<T> NoticiaPublicada;

    public async Task PublicarVariasAsync(List<Func<Task<T>>> generadores)
    {
        List<Task<T>> tareas = new List<Task<T>>();

        // Lanzamos todos los generadores en paralelo (sin await aún)
        foreach (var generador in generadores)
        {
            tareas.Add(generador());
        }

        // Esperamos a que TODOS terminen
        T[] noticias = await Task.WhenAll(tareas);

        // Publicamos cada noticia a los suscriptores
        foreach (T noticia in noticias)
        {
            NoticiaPublicada?.Invoke(noticia);
        }
    }
}

class Programa
{
    // Generador async: "obtiene" una noticia tardando un tiempo
    static async Task<string> NoticiaDeportes()
    {
        await Task.Delay(1500);
    }
}
```

```

        return "Deportes: el equipo local ganó 3-1.";
    }

    static async Task<string> NoticiaClima()
    {
        await Task.Delay(800);
        return "Clima: soleado con 28 grados.";
    }

    static async Task<string> NoticiaTecnologia()
    {
        await Task.Delay(1200);
        return "Tecnología: nuevo lenguaje de programación causa furor.";
    }

    static async Task Main(string[] args)
    {
        Noticiero<string> noticiero = new Noticiero<string>();

        // Suscriptor: imprime cada noticia con la hora
        noticiero.NoticiaPublicada += noticia =>
        {
            Console.WriteLine "[" + DateTime.Now.ToString("HH:mm:ss") + "] " +
noticia);
        };

        // Los generadores como delegados Func<Task<string>>
        List<Func<Task<string>>> generadores = new List<Func<Task<string>>>
        {
            () => NoticiaDeportes(),
            () => NoticiaClima(),
            () => NoticiaTecnologia()
        };

        Console.WriteLine("Recopilando noticias en paralelo...");
        await noticiero.PublicarVariasAsync(generadores);
        Console.WriteLine("Noticiero terminado.");
    }
}

```

Explicación: los tres generadores arrancan a la vez y cada uno espera su propio tiempo sin bloquear; el total es ~1.5 segundos (el más lento) en lugar de 3.5. Cuando todos terminan, el noticiero publica cada noticia mediante el evento y el suscriptor la imprime con la hora. Cambia T a otro tipo (por ejemplo, una clase Noticia con título y categoría) y todo sigue funcionando sin tocar la clase Noticiero: esa es la reutilización que dan los genéricos, con la flexibilidad de los

delegados y la eficiencia de async. Si llegaste hasta aquí y lo entiendes, ya piensas como programador profesional de C#.

Resumen de la unidad

- async y await permiten esperar operaciones lentas sin bloquear el hilo: el programa sigue respondiendo mientras la espera ocurre en segundo plano.
- Task representa una operación en curso (una promesa de resultado futuro); Task<T> además entrega un valor cuando termina.
- await pausa el método sin bloquear, libera el hilo para otro trabajo y retoma donde se quedó cuando la operación termina.
- Task.Run ejecuta trabajo pesado de CPU en un hilo del pool; no lo uses para operaciones que ya tienen versión asíncrona (red, disco, base de datos).
- Task.Delay es la espera asíncrona (reemplazo de Thread.Sleep); Task.WhenAll espera a que varias tareas terminen juntas, permitiendo paralelizar esperas.
- Errores clásicos: async void (solo para eventos), olvidar el await, mezclar .Result/.Wait() con async (deadlocks), y creer que async acelera una operación individual.
- Buenas prácticas: "async all the way" en toda la cadena de llamadas, sufijo Async en los métodos, y evitar async void fuera de eventos.
- Un delegado es una variable que guarda un método; permite pasar comportamiento como parámetro, como hace LINQ con Where y Select.
- Func<T> devuelve un valor, Action<T> no devuelve nada y Predicate<T> devuelve bool: son los delegados genéricos que usa todo .NET.
- Las lambdas (=>) crean delegados en una línea y pueden capturar variables del contexto (closures); cuidado con variables de ciclo que cambian.
- Los eventos implementan el patrón publicador-suscriptor: una clase publica avisos y otras se suscriben con += sin acoplarse entre sí.
- Dispara eventos con ?.Invoke para no fallar cuando no hay suscriptores, y usa la convención EventHandler<TEventArgs> con (sender, e).
- Los genéricos (List<T>, clases y métodos con <T>) permiten escribir código una vez y usarlo con cualquier tipo, con seguridad en compilación.
- Las restricciones where T (class, struct, new(), interfaces) le prometen al compilador qué puede hacer T a cambio de permitir esas operaciones.

- `List<T>` supera a `ArrayList`: sin boxing/unboxing, sin casts, errores detectados al compilar y mejor rendimiento.
- La combinación de `async` + delegados + genéricos es la base del código profesional en C#: reutilizable, desacoplado y que no se congela.

UNIDAD 10: Proyectos finales

Llegaste a la última unidad del Curso Completo de C#. Tómate un segundo para mirar atrás: empezaste sin saber qué era una variable y ahora manejas clases, herencia, colecciones, LINQ, archivos, asincronía, delegados y genéricos. Eso no es poco: es el temario completo de un curso universitario de programación, comprimido en diez unidades. Pero hay una diferencia entre "saber los temas" y "saber programar", y esa diferencia se llama proyectos.

Piénsalo así: puedes estudiar todas las reglas del fútbol, memorizar las posiciones y ver cien partidos por televisión, pero hasta que no entras a la cancha con un balón, no sabes jugar. Con la programación pasa igual. Los ejercicios pequeños de cada unidad te enseñaron las jugadas; los proyectos de esta unidad te meten a la cancha. Vas a construir dos programas completos, de principio a fin, como los haría un programador profesional: con menú, manejo de errores, código organizado en métodos y persistencia de datos.

El primer proyecto es una calculadora de consola completa. Parece sencillo, y esa es la trampa: una calculadora "de verdad" no es solo sumar dos números. Tiene un menú que se repite hasta que el usuario decide salir, valida que lo que el usuario escribió sean números de verdad, maneja la división entre cero sin tumbarse, permite operaciones encadenadas y está organizada en métodos con una sola responsabilidad cada uno. Al construirla vas a repasar ciclos, decisiones, métodos, excepciones y refactorización, todo junto en un programa que sí usarías.

El segundo proyecto es una agenda de contactos con persistencia en archivos. Aquí subes de nivel: los datos ya no viven solo en la memoria mientras el programa corre, sino que se guardan en un archivo de texto en el disco y se recuperan al abrir el programa. Vas a implementar agregar, listar, buscar y eliminar contactos, y el menú completo que los une. Este proyecto te enseña el ciclo de vida real de los datos en una aplicación: crear, leer, actualizar, eliminar (lo que en la industria se llama CRUD, por sus siglas en inglés: Create, Read, Update, Delete). Casi todas las aplicaciones del mundo, desde la app de tu banco hasta una red social, son un CRUD con mejor ropa.

La tercera parte de la unidad, la clase 30, es diferente: no es código, es brújula. Te muestra el panorama de lo que sigue después de este curso dentro del ecosistema .NET: ASP.NET Core para crear APIs y aplicaciones web del lado del servidor, .NET MAUI para aplicaciones móviles y de escritorio con un solo código base, Unity para videojuegos (el motor con el que se hicieron juegos que seguro has jugado), y Blazor para aplicaciones web interactivas escritas en C# en lugar de JavaScript. Además, verás cómo seguir aprendiendo por tu cuenta, cómo armar un portafolio que consiga entrevistas y consejos directos para buscar tu primer trabajo como programador.

¿Por qué importa todo esto en la vida real? Porque nadie contrata a alguien por "saber la sintaxis de C#". Te contratan por poder construir cosas que funcionen. Cuando vayas a una entrevista, no te van a pedir que recites la definición de polimorfismo: te van a pedir que muestres qué has construido. Estos dos proyectos son las primeras dos piezas de tu portafolio, y la guía de carrera te dice cómo conseguir las siguientes. Al terminar esta unidad no solo sabrás C#: tendrás programas terminados que lo demuestran y un mapa claro de a dónde ir después.

Un consejo antes de empezar: escribe el código tú mismo, no solo lo leas. Copiar y pegar no enseña; los errores que cometes al escribir (y que luego corrigas) son los que fijan el conocimiento. Cada proyecto está construido por etapas: primero una versión mínima que funciona, luego se le agregan piezas. Esa es la forma profesional de trabajar: nadie escribe un programa grande de una sentada; se construye por capas, probando cada una.

Explicación detallada

Cómo se construye un proyecto real: piensa antes de codificar

Los principiantes suelen abrir el editor y empezar a escribir código inmediatamente. Los profesionales hacen algo antes: piensan. No se trata de hacer diagramas complicados, sino de responder tres preguntas en una hoja o en un comentario: ¿qué debe hacer el programa? (los requisitos), ¿qué partes tiene? (el diseño), ¿en qué orden las construyo? (el plan).

Para la calculadora, los requisitos son: mostrar un menú con operaciones (suma, resta, multiplicación, división), pedir los números al usuario, mostrar el resultado, repetir hasta que el usuario elija salir, no tumbarse con entradas inválidas ni con división entre cero, y tener el código organizado en métodos. El diseño: un método por operación, un método para leer números validados, un método para mostrar el menú y un Main que orquesta el ciclo. El plan: primero la versión mínima (una suma), luego el menú con todas las operaciones, luego el manejo de errores, luego la refactorización.

Para la agenda, los requisitos son: agregar contactos (nombre, teléfono, correo), listar todos, buscar por nombre, eliminar por nombre, guardar en archivo al salir (o al cada cambio) y cargar al iniciar. El diseño: una clase Contacto, una clase Agenda que maneja la lista y el archivo, y un Main con el menú. El plan: primero el modelo y las operaciones en memoria, luego el menú, luego la persistencia en archivo.

Esta forma de trabajar tiene un nombre elegante en la industria (desarrollo incremental), pero la idea es de sentido común: construye algo pequeño que funcione, pruébalo, y luego agrándalo. Si algo se rompe, sabes que fue lo último que agregaste. Es mil veces mejor que escribir 300 líneas de un jalón y luego pasar dos horas buscando dónde está el error.

Diseño de la calculadora: métodos con una sola responsabilidad

Un buen diseño divide el problema en piezas pequeñas, cada una con un trabajo claro. Para la calculadora, la división natural es:

- `MostrarMenu()`: imprime las opciones y devuelve la opción elegida como texto.
- `LeerNumero(mensaje)`: pide un número al usuario y no regresa hasta recibir uno válido (usa `double.TryParse` en un ciclo, sin excepciones para el flujo normal).
- `Sumar(a, b)`, `Restar(a, b)`, `Multiplicar(a, b)`, `Dividir(a, b)`: las operaciones puras, que solo calculan.
- `Main()`: el ciclo principal que muestra el menú, pide los datos según la opción y muestra resultados.

Observa la separación: los métodos de operación no saben nada de la consola; solo reciben números y devuelven números. Eso los hace fáciles de probar y de reutilizar. El método `LeerNumero` concentra toda la validación en un solo lugar: si mañana quieres cambiar el mensaje de error, lo cambias una vez. Y `Main` se dedica a orquestar, no a calcular. Este principio (cada método hace una cosa y la hace bien) se llama responsabilidad única, y es la base del código mantenible.

Sobre la división entre cero: con `double`, dividir entre cero no lanza excepción en C#, sino que devuelve Infinito (`Infinity`). Eso es matemáticamente discutible pero no tumba el programa. Aun así, en nuestra calculadora vamos a detectarlo y mostrar un mensaje amable, porque "Infinito" confunde al usuario normal. Es un ejemplo de que manejar errores no es solo evitar que el programa se caiga: es cuidar la experiencia de quien lo usa.

Diseño de la agenda: el patrón CRUD y la persistencia

La agenda introduce un concepto central: la persistencia. Los datos en memoria (una `List<Contacto>`) desaparecen cuando el programa termina. Para que los contactos sigan ahí mañana, hay que guardarlos en un archivo. El formato más simple es texto plano con un contacto por línea y los campos separados por un carácter especial, por ejemplo el punto y coma: "Ana Torres;5551234567;ana@correo.com". Este formato se llama CSV (valores separados por comas, aunque aquí usemos punto y coma para evitar conflictos) y lo abren Excel y casi todo.

El flujo de la agenda es: al iniciar, si el archivo existe, leerlo línea por línea, partir cada línea por el separador y reconstruir los contactos en la lista. Durante el uso, todas las operaciones (agregar, buscar, eliminar) trabajan sobre la lista en memoria, que es rápida. Al salir (o después de cada cambio, según el diseño), escribir toda la lista de vuelta al archivo. Este ciclo "cargar al inicio, trabajar en memoria, guardar al final" es el que usan miles de aplicaciones pequeñas reales.

La clase `Contacto` es un ejemplo perfecto de lo que aprendiste en la unidad de POO: propiedades, constructor y un método `ToString` para mostrarse bonito. La clase `Agenda` encapsula la lista y las operaciones sobre ella, incluyendo `GuardarEnArchivo` y `CargarDeArchivo`. El `Main` solo maneja el menú y la interacción. Fíjate cómo cada clase tiene su responsabilidad: `Contacto` guarda datos, `Agenda` maneja la colección y el archivo, `Main` habla con el usuario.

Un detalle importante de diseño: ¿qué pasa si el usuario agrega un contacto con punto y coma en el nombre? Rompería el formato. En una aplicación profesional se usaría un formato más robusto (JSON, por ejemplo, que verás en el siguiente nivel con ASP.NET Core). Para este proyecto, validamos que los campos no contengan el separador, y lo documentamos como limitación conocida. Reconocer las limitaciones de tu diseño es una habilidad profesional: ningún programa es perfecto, pero un buen programador sabe dónde están sus bordes.

Archivos de texto en C#: lo esencial para la agenda

Ya viste archivos en la unidad correspondiente, pero hagamos un repaso enfocado en lo que la agenda necesita. Para escribir, `File.WriteAllLines(ruta, lineas)` escribe un arreglo o lista de cadenas, una por línea, creando o reemplazando el archivo. Para leer, `File.ReadAllLines(ruta)` devuelve un arreglo con cada línea, y `File.Exists(ruta)` verifica si el archivo existe antes de intentar leerlo (si no existe, simplemente empezamos con agenda vacía: no es un error, es la primera vez que se usa el programa).

Para convertir un Contacto a línea de texto y de vuelta, usamos `string.Join(";", partes)` para unir y `linea.Split(';')` para partir. Cada operación es de una línea, pero juntas forman la persistencia completa. Envuelve la lectura y escritura en `try/catch` por si el archivo está bloqueado o no hay permisos: un programa que se cae al guardar es un programa que pierde datos del usuario, y eso es imperdonable.

Guía de carrera: el mapa después del curso

Ahora la parte de brújula. C# no es solo un lenguaje: es la puerta a un ecosistema enorme llamado .NET, y cada rama lleva a un tipo distinto de trabajo. Conocer el mapa te permite elegir tu siguiente paso con intención en lugar de dar vueltas.

ASP.NET Core es el framework para construir aplicaciones web del lado del servidor y APIs (las interfaces que usan las apps móviles y los sitios web para obtener datos). Si te gusta la idea de construir el "cerebro" detrás de las aplicaciones (servidores, bases de datos, lógica de negocio), este es tu camino. Es uno de los perfiles mejor pagados: casi toda empresa mediana o grande necesita desarrolladores backend, y .NET es fuerte en bancos, gobierno y empresas grandes. El camino de aprendizaje sería: C# sólido (lo que ya tienes), luego HTTP y APIs REST, luego ASP.NET Core con Entity Framework para bases de datos, y finalmente despliegue en la nube (Azure es el primo hermano de .NET).

.NET MAUI es el framework para crear aplicaciones móviles (Android, iOS) y de escritorio (Windows, Mac) con un solo código base en C#. Si te imaginas publicando tu propia app en la Play Store, este es el camino. La ventaja es obvia: aprendes una vez y llegas a todas las plataformas, en lugar de aprender Kotlin para Android y Swift para iPhone por separado.

Unity es el motor de videojuegos más popular del mundo para desarrolladores independientes, y su lenguaje de scripting es C#. Juegos famosos hechos en Unity hay por cientos, desde indies queridos hasta títulos móviles que generan millones. Si los videojuegos son tu pasión, tu C# ya te puso a medio camino: lo que sigue es aprender el editor de Unity, su sistema de físicas y su ciclo de vida de objetos. Es un camino más competitivo pero también el más divertido para muchos.

Blazor permite construir aplicaciones web interactivas usando C# en lugar de JavaScript, ejecutándose en el navegador mediante WebAssembly. Si te gusta la web pero prefieres quedarte en el mundo C#, Blazor es tu puente: reutilizas tu conocimiento del lenguaje para el frontend.

¿Cómo seguir aprendiendo? Con proyectos, siempre con proyectos. La regla de oro: por cada hora de tutorial, dos horas de construir algo propio. Los tutoriales dan la ilusión de saber ("lo entiendo cuando lo veo"), pero el conocimiento real solo llega cuando te atorras tú solo y lo

resuelves. Elige un tema que te emocione (una app para tu negocio, un juego sencillo, una API de tus películas favoritas) y constrúyelo; cada proyecto te obligará a aprender justo lo que necesitas.

Tu portafolio es tu currículum real. Crea una cuenta en GitHub (es gratis) y sube tus proyectos: la calculadora, la agenda y los que hagas después. Cada repositorio debe tener un archivo README que explique qué hace el proyecto, cómo ejecutarlo y qué aprendiste haciéndolo. Los reclutadores revisan GitHub: un perfil con proyectos terminados y código limpio dice más que cualquier certificado. Consejo práctico: mejor tres proyectos pequeños terminados y pulidos que diez a medias.

Para conseguir trabajo, además del portafolio: aprende Git bien (es el sistema de control de versiones que usan todos los equipos; saber crear ramas y resolver conflictos es requisito en casi cualquier vacante), practica explicar tu código en voz alta (en las entrevistas te pedirán que describas lo que hiciste y por qué), y no tengas miedo de aplicar a puestos "junior" aunque sientas que no cumples el 100% de los requisitos: las listas de requisitos describen al candidato ideal, no al mínimo. Y un consejo mexicano y directo: en este gremio nadie te va a regalar nada, pero el que construye, muestra y no se rinde, encuentra su lugar. Ya tienes la base; lo que sigue es constancia.

Errores comunes en proyectos (y cómo evitarlos)

En proyectos reales aparecen errores que los ejercicios pequeños no muestran. El primero es el "código espagueti": todo en el Main, con ciclos anidados y variables que se usan en diez lugares. Se evita con la refactorización que practicarás en la calculadora: extraer métodos con nombres claros hasta que Main se lea como una receta.

El segundo es no validar entradas. Todo lo que escribe el usuario es sospechoso hasta demostrar lo contrario: texto donde van números, opciones fuera de rango, campos vacíos. La agenda y la calculadora validan todo, y tus futuros proyectos deben hacerlo también. La validación no es desconfianza: es profesionalismo.

El tercero es perder datos por no manejar errores de archivo. Siempre envuelve la lectura/escritura en try/catch, y en programas más serios, guarda respaldos o escribe primero a un archivo temporal. En nuestra agenda, si falla el guardado, avisamos al usuario en lugar de fingir que todo salió bien.

El cuarto es el alcance descontrolado ("ya que estoy, le agrego esto y esto otro..."). Termina primero la versión mínima funcional y luego agrega extras. Un proyecto terminado y simple vale más que uno ambicioso a medias. Los ejercicios de práctica te proponen extensiones justo con esa filosofía: primero que funcione, luego que brille.

Depurar como profesional: encontrar errores en proyectos grandes

En los ejercicios pequeños, cuando algo fallaba, el error estaba a la vista. En proyectos de 300 líneas como la agenda, un error puede esconderse en cualquier parte. Depurar (encontrar y corregir errores) es una habilidad tan importante como escribir código, y tiene técnica.

La técnica más simple y más usada se llama "depuración con impresiones": pones `Console.WriteLine` en puntos estratégicos para ver qué está pasando. ¿La lista llega vacía a Guardar? Imprime `contacts.Count` antes de guardar. ¿El archivo se lee pero los contactos salen raros? Imprime cada línea justo después de leerla. No es elegante, pero funciona siempre y los profesionales la usan a diario para errores rápidos.

La técnica seria es usar el depurador del editor (Visual Studio o VS Code): pones un "punto de interrupción" (breakpoint) en una línea, ejecutas, y el programa se detiene ahí para que inspecciones el valor de cada variable paso a paso. Aprender a usar breakpoints te ahorra horas: en lugar de adivinar qué vale una variable, lo ves con tus ojos. Si tu editor lo soporta, practica con la agenda: pon un breakpoint en `BuscarPorNombre` y mira cómo cambia la lista de encontrados en cada vuelta del ciclo.

Tres reglas de oro para depurar. Primera: reproduce el error de forma confiable antes de intentar arreglarlo. Si no sabes cómo provocar el error a voluntad, no sabrás si tu arreglo funcionó. Segunda: cambia una sola cosa a la vez. Si cambias tres cosas y el error desaparece, no sabes cuál lo arregló (y puede que solo lo hayas escondido). Tercera: lee el mensaje de excepción completo, incluyendo el stack trace (la lista de métodos que muestra dónde ocurrió). Los principiantes ven `"NullReferenceException"` y entran en pánico; los profesionales leen la línea exacta que indica y van directo ahí. El 90% de las veces, el mensaje ya te dice qué pasó y dónde.

Un hábito que te va a servir toda la carrera: prueba cada método por separado con casos extremos. ¿Qué hace `Desencolar` con la cola vacía? ¿Qué hace `Dividir` con `b = 0`? ¿Qué hace `BuscarPorNombre` con texto vacío? ¿Qué pasa si el archivo `agenda.txt` está vacío o tiene una línea corrupta? Los programadores profesionales piensan en estos casos bordes antes de que el usuario los encuentre. Cada validación que agregaste en estos proyectos nació de preguntarte "¿y si el usuario hace esto?".

De la consola al mundo real: a dónde llevan estos proyectos

Tus dos proyectos son de consola, pero los patrones que implementan son los mismos que usan las aplicaciones grandes. Ver la conexión te ayuda a elegir tu siguiente paso después del curso.

La calculadora es, en el fondo, lógica de negocio pura separada de la interfaz: los métodos Sumar, Dividir y RaizCuadrada no saben nada de la consola. Esa separación es exactamente lo que pide ASP.NET Core: la misma lógica, pero en lugar de leer de Console.ReadLine, recibe peticiones HTTP y devuelve respuestas JSON. Si mañana conviertes tu calculadora en una API web, los métodos de cálculo se copian tal cual; solo cambia la "capa de presentación". Por eso insistimos tanto en no mezclar cálculo con consola: el código limpio migra, el código espagueti se queda atorado.

La agenda es un CRUD con persistencia, que es el esqueleto del 80% de las aplicaciones empresariales. En ASP.NET Core, la List<Contacto> se reemplazaría por una base de datos real (SQL Server o SQLite) accedida con Entity Framework, y el menú de consola por endpoints de API (GET /contactos, POST /contactos, DELETE /contactos/{nombre}). En .NET MAUI, la misma clase Contacto y la misma lógica de Agenda vivirían en una app móvil con botones y listas táctiles en lugar de menú de texto. El archivo agenda.txt sería una base de datos local en el teléfono. ¿Ves el patrón? Lo que cambia entre plataformas es la interfaz y el almacenamiento; la lógica (validar, buscar, eliminar) es la misma que ya escribiste. Por eso estos proyectos son inversión, no tarea escolar: cada línea te sirve en el siguiente nivel.

Incluso el inventario del ejercicio 7 tiene un primo famoso en el mundo real: los sistemas punto de venta de las tienditas y los ERP de las empresas hacen exactamente Vender, Surtir y reportes de valor de inventario, solo que con códigos de barras, tickets impresos y bases de datos. Cuando en una entrevista cuentas que construiste un inventario con persistencia y validaciones, estarás describiendo experiencia relevante, no un ejercicio.

Ejemplos paso a paso

Proyecto 1: Calculadora de consola, etapa por etapa

Vamos a construir la calculadora como lo haría un profesional: por etapas, cada una funcional y probada antes de pasar a la siguiente. Te recomiendo escribir cada etapa en tu editor, ejecutarla y jugar con ella antes de continuar.

Etapa 1: la versión mínima. Solo suma dos números. El objetivo es tener algo que funcione de inmediato, aunque sea básico.

```

using System;

class Programa
{
    static void Main()
    {
        // Etapa 1: lo mínimo que se puede llamar "calculadora"
        Console.Write("Primer número: ");
        double a = double.Parse(Console.ReadLine());

        Console.Write("Segundo número: ");
        double b = double.Parse(Console.ReadLine());

        double resultado = a + b;
        Console.WriteLine("Resultado: " + resultado);
    }
}

```

Funciona, pero es frágil: si escribes "hola" en lugar de un número, el programa se cae con una excepción. Y solo suma una vez. Lo anotamos como deuda técnica y seguimos: en la etapa 3 lo arreglaremos. La filosofía es avanzar con algo funcional y pulir después, no buscar la perfección en el primer intento.

Etapa 2: menú con las cuatro operaciones y ciclo de repetición. Ahora el programa no termina después de una operación: muestra un menú y repite hasta que el usuario elija salir.

```

using System;

class Programa
{
    static void Main()
    {
        // Etapa 2: menú repetitivo con las 4 operaciones básicas
        bool salir = false;

        while (!salir)
        {
            // Mostrar el menú
            Console.WriteLine();
            Console.WriteLine("=== CALCULADORA ===");
            Console.WriteLine("1. Sumar");
            Console.WriteLine("2. Restar");
            Console.WriteLine("3. Multiplicar");
            Console.WriteLine("4. Dividir");

```

```

Console.WriteLine("0. Salir");
Console.Write("Elige una opción: ");
string opcion = Console.ReadLine();

// La opción 0 sale del ciclo
if (opcion == "0")
{
    salir = true;
    Console.WriteLine("¡Hasta luego!");
    continue; // Vuelve al while, que ya no se repite
}

// Pedir los dos números (todavía sin validación: etapa 3)
Console.Write("Primer número: ");
double a = double.Parse(Console.ReadLine());
Console.Write("Segundo número: ");
double b = double.Parse(Console.ReadLine());

double resultado = 0;
bool operacionValida = true;

// switch para elegir la operación según la opción
switch (opcion)
{
    case "1":
        resultado = a + b;
        break;
    case "2":
        resultado = a - b;
        break;
    case "3":
        resultado = a * b;
        break;
    case "4":
        resultado = a / b; // División entre cero: lo veremos en etapa 3
        break;
    default:
        Console.WriteLine("Opción no válida.");
        operacionValida = false;
        break;
}

if (operacionValida)
{
    Console.WriteLine("Resultado: " + resultado);
}

```

```

    }
}
}
}

```

Ya parece un programa de verdad: menú, ciclo, cuatro operaciones. Pero todo está en Main, la validación brilla por su ausencia y la división entre cero muestra "Infinito". Es hora de la etapa 3.

Etapa 3: validación y manejo de errores. Agregamos un método LeerNumero que no acepta basura, y controlamos la división entre cero con un mensaje claro. También validamos la opción del menú.

```

using System;

class Programa
{
    // Lee un número validado: repite hasta recibir un double válido
    static double LeerNumero(string mensaje)
    {
        double numero;
        while (true)
        {
            Console.Write(mensaje);
            string entrada = Console.ReadLine();

            // TryParse no lanza excepción: devuelve true/false
            if (double.TryParse(entrada, out numero))
            {
                return numero; // Entrada válida: salimos con el valor
            }
            Console.WriteLine("Eso no es un número válido. Intenta de nuevo.");
        }
    }

    static void Main()
    {
        bool salir = false;

        while (!salir)
        {
            Console.WriteLine();
            Console.WriteLine("=== CALCULADORA ===");
            Console.WriteLine("1. Sumar");
            Console.WriteLine("2. Restar");

```



```

Console.WriteLine("3. Multiplicar");
Console.WriteLine("4. Dividir");
Console.WriteLine("0. Salir");
Console.Write("Elige una opción: ");
string opcion = Console.ReadLine();

if (opcion == "0")
{
    salir = true;
    Console.WriteLine("¡Hasta luego!");
    continue;
}

// Validar que la opción esté entre 1 y 4
if (opcion != "1" && opcion != "2" && opcion != "3" && opcion != "4")
{
    Console.WriteLine("Opción no válida. Elige entre 0 y 4.");
    continue; // Vuelve al inicio del ciclo sin pedir números
}

// Ahora los números siempre son válidos gracias a LeerNumero
double a = LeerNumero("Primer número: ");
double b = LeerNumero("Segundo número: ");

double resultado = 0;
string simbolo = "";

switch (opcion)
{
    case "1":
        resultado = a + b; simbolo = "+";
        break;
    case "2":
        resultado = a - b; simbolo = "-";
        break;
    case "3":
        resultado = a * b; simbolo = "*";
        break;
    case "4":
        // División entre cero: mensaje amable en lugar de Infinito
        if (b == 0)
        {
            Console.WriteLine("Error: no se puede dividir entre cero.");
            continue;
        }
}

```

```

        resultado = a / b; simbolo = "/";
        break;
    }

    // Muestra la operación completa: "5 + 3 = 8"
    Console.WriteLine(a + " " + simbolo + " " + b + " = " + resultado);
}
}
}

```

El programa ya es robusto: ninguna entrada del usuario puede tumbarlo. El truco de LeerNumero (ciclo infinito con return al validar) es un patrón que usarás en todos tus programas de consola. Y el continue después de cada error evita anidar if tras if: en lugar de "si todo está bien, haz esto dentro de diez llaves", decimos "si algo está mal, avisa y vuelve a empezar".

Etapa 4: refactorización final. El código funciona, pero Main está largo y mezcla responsabilidades: muestra menú, valida, calcula, imprime. La refactorización extrae métodos con una sola responsabilidad hasta que Main se lea como una receta de cocina. Este es el código completo y final de la calculadora, comentado línea por línea:

```

using System;

class Calculadora
{
    // Muestra el menú y devuelve la opción elegida (ya validada)
    static string MostrarMenu()
    {
        Console.WriteLine();
        Console.WriteLine("=== CALCULADORA ===");
        Console.WriteLine("1. Sumar");
        Console.WriteLine("2. Restar");
        Console.WriteLine("3. Multiplicar");
        Console.WriteLine("4. Dividir");
        Console.WriteLine("5. Potencia");
        Console.WriteLine("6. Raíz cuadrada");
        Console.WriteLine("0. Salir");

        // Ciclo de validación: no regresa hasta recibir opción válida
        while (true)
        {
            Console.Write("Elige una opción: ");
            string opcion = Console.ReadLine();
            if (opcion == "0" || opcion == "1" || opcion == "2" ||
                opcion == "3" || opcion == "4" || opcion == "5" ||

```

```

        opcion == "6")
    {
        return opcion;
    }
    Console.WriteLine("Opción no válida. Intenta de nuevo.");
}
}

// Lee un número validado del usuario (repite hasta que sea válido)
static double LeerNumero(string mensaje)
{
    while (true)
    {
        Console.Write(mensaje);
        double numero;
        if (double.TryParse(Console.ReadLine(), out numero))
        {
            return numero;
        }
        Console.WriteLine("Eso no es un número válido. Intenta de nuevo.");
    }
}

// Las operaciones puras: solo calculan, no saben nada de la consola.
// Al estar separadas, se pueden probar y reutilizar fácilmente.
static double Sumar(double a, double b) { return a + b; }
static double Restar(double a, double b) { return a - b; }
static double Multiplicar(double a, double b) { return a * b; }

static double Dividir(double a, double b)
{
    // La validación vive aquí, junto a la operación que la necesita
    if (b == 0)
    {
        throw new DivideByZeroException("No se puede dividir entre cero.");
    }
    return a / b;
}

static double Potencia(double a, double b) { return Math.Pow(a, b); }

static double RaizCuadrada(double a)
{
    if (a < 0)
    {

```

```

        throw new ArgumentException("No hay raíz real de un número negativo.");
    }
    return Math.Sqrt(a);
}

// Ejecuta la operación elegida: pide datos, calcula y muestra
static void EjecutarOperacion(string opcion)
{
    try
    {
        double a, b, resultado;

        switch (opcion)
        {
            case "1":
                a = LeerNumero("Primer número: ");
                b = LeerNumero("Segundo número: ");
                resultado = Sumar(a, b);
                Console.WriteLine(a + " + " + b + " = " + resultado);
                break;

            case "2":
                a = LeerNumero("Primer número: ");
                b = LeerNumero("Segundo número: ");
                resultado = Restar(a, b);
                Console.WriteLine(a + " - " + b + " = " + resultado);
                break;

            case "3":
                a = LeerNumero("Primer número: ");
                b = LeerNumero("Segundo número: ");
                resultado = Multiplicar(a, b);
                Console.WriteLine(a + " * " + b + " = " + resultado);
                break;

            case "4":
                a = LeerNumero("Primer número: ");
                b = LeerNumero("Segundo número: ");
                resultado = Dividir(a, b); // Puede lanzar DivideByZeroException
                Console.WriteLine(a + " / " + b + " = " + resultado);
                break;

            case "5":
                a = LeerNumero("Base: ");
                b = LeerNumero("Exponente: ");

```

```

        resultado = Potencia(a, b);
        Console.WriteLine(a + " ^ " + b + " = " + resultado);
        break;

    case "6":
        a = LeerNumero("Número: ");
        resultado = RaizCuadrada(a); // Puede lanzar ArgumentException
        Console.WriteLine("Raíz de " + a + " = " + resultado);
        break;
    }
}
catch (Exception ex)
{
    // Un solo catch centraliza los errores de cálculo:
    // el programa nunca se cae, siempre explica qué pasó
    Console.WriteLine("Error: " + ex.Message);
}
}

static void Main()
{
    // Main ahora es una receta legible: menú, ejecutar, repetir
    Console.WriteLine("Bienvenido a la calculadora de Lucas EGA Academy.");

    while (true)
    {
        string opcion = MostrarMenu();

        if (opcion == "0")
        {
            Console.WriteLine("¡Gracias por usar la calculadora!");
            break;
        }

        EjecutarOperacion(opcion);
    }
}
}

```

Compara este Main con el de la etapa 2: de 60 líneas mezcladas a 12 líneas que cuentan una historia clara. Esa es la refactorización: el programa hace lo mismo, pero el código es más fácil de leer, de probar y de extender. Agregar una operación nueva ahora es trivial: un método puro, un case en el switch y una línea en el menú. Cuando en una entrevista te pidan "agrega el módulo a tu calculadora", esto es lo que quieren ver.

Nota el diseño de errores en dos niveles: LeerNumero evita excepciones para entradas inválidas (usa TryParse, porque un usuario que escribe mal no es un caso "excepcional", es lo normal), mientras que Dividir y RaizCuadrada lanzan excepciones para condiciones matemáticamente inválidas, atrapadas en un solo catch en EjecutarOperacion. Esa distinción (validar lo esperado con if/TryParse, excepciones para lo excepcional) es criterio de programador experimentado.

Proyecto 2: Agenda de contactos con archivos, etapa por etapa

La agenda es un paso más allá: introduce la persistencia. La construiremos también por etapas: primero el modelo de datos y las operaciones en memoria, luego el menú, y al final el guardado en archivo.

Etapas 1: la clase Contacto y operaciones en memoria. Sin menú ni archivos todavía: solo el modelo y la lógica, probados directamente en Main.

```
using System;
using System.Collections.Generic;

// Etapa 1: el modelo de datos. Un contacto es nombre + teléfono + correo.
class Contacto
{
    // Propiedades automáticas: la forma moderna de guardar datos
    public string Nombre { get; set; }
    public string Telefono { get; set; }
    public string Correo { get; set; }

    public Contacto(string nombre, string telefono, string correo)
    {
        Nombre = nombre;
        Telefono = telefono;
        Correo = correo;
    }

    // ToString: cómo se muestra un contacto al imprimirlo
    public override string ToString()
    {
        return Nombre + " | Tel: " + Telefono + " | Correo: " + Correo;
    }

    // Convierte el contacto a una línea de texto para el archivo
    public string ALinea()
    {
        return Nombre + ";" + Telefono + ";" + Correo;
    }
}
```

```

    }

    // Reconstruye un contacto desde una línea del archivo (método static)
    public static Contacto DesdeLinea(string linea)
    {
        string[] partes = linea.Split(';');
        // Si la línea viene corrupta, devolvemos null y el llamador decide
        if (partes.Length != 3)
        {
            return null;
        }
        return new Contacto(partes[0], partes[1], partes[2]);
    }
}

class Programa
{
    static void Main()
    {
        // Prueba rápida de la etapa 1: crear y convertir contactos
        Contacto c = new Contacto("Ana Torres", "5551234567", "ana@correo.com");
        Console.WriteLine(c); // Usa ToString automáticamente

        string linea = c.ALinea();
        Console.WriteLine("En archivo se guarda así: " + linea);

        Contacto recuperado = Contacto.DesdeLinea(linea);
        Console.WriteLine("Recuperado: " + recuperado);
    }
}

```

Fíjate en el diseño: Contacto sabe convertirse a línea y reconstruirse desde línea, pero no sabe nada de archivos. Esa responsabilidad será de la clase Agenda. Cada clase en su carril.

Etapla 2: la clase Agenda con el CRUD en memoria. Agregar, listar, buscar y eliminar, todo sobre una `List<Contacto>` privada.

```

using System;
using System.Collections.Generic;

class Agenda
{
    // La lista es privada: solo los métodos de Agenda pueden modificarla.
    // Esto se llama encapsulamiento y evita que alguien la corrompa desde fuera.
    private List<Contacto> contactos = new List<Contacto>();
}

```

```

// CREATE: agregar un contacto nuevo
public void Agregar(Contacto contacto)
{
    contactos.Add(contacto);
}

// READ: obtener todos los contactos (copia de solo lectura)
public List<Contacto> ObtenerTodos()
{
    return new List<Contacto>(contactos);
}

// READ: buscar por nombre (búsqueda parcial, sin importar mayúsculas)
public List<Contacto> BuscarPorNombre(string texto)
{
    List<Contacto> encontrados = new List<Contacto>();
    foreach (Contacto c in contactos)
    {
        // IndexOf con IgnoreCase: encuentra "ana" dentro de "Ana Torres"
        if (c.Nombre.IndexOf(texto, StringComparison.OrdinalIgnoreCase) >= 0)
        {
            encontrados.Add(c);
        }
    }
    return encontrados;
}

// DELETE: eliminar por nombre exacto; devuelve true si lo encontró
public bool Eliminar(string nombre)
{
    for (int i = 0; i < contactos.Count; i++)
    {
        if (contactos[i].Nombre.Equals(nombre,
StringComparison.OrdinalIgnoreCase))
        {
            contactos.RemoveAt(i);
            return true; // Lo encontramos y eliminamos
        }
    }
    return false; // No existía ningún contacto con ese nombre
}

public int Cantidad
{
    get { return contactos.Count; }
}

```



```
}  
}
```

Nota las decisiones: `BuscarPorNombre` hace búsqueda parcial e insensible a mayúsculas (así "ana" encuentra "Ana Torres"), porque eso es lo que un usuario espera. `Eliminar` usa nombre exacto (insensible a mayúsculas) para no borrar por accidente algo que solo se parecía. `ObtenerTodos` devuelve una copia para que nadie modifique la lista interna por accidente. Son detalles pequeños que separan un programa "que funciona" de uno "bien hecho".

Etapa 3: persistencia en archivo. Agregamos a `Agenda` los métodos `Guardar` y `Cargar`. El archivo se llama "agenda.txt" y vive junto al programa.

```
using System;  
using System.Collections.Generic;  
using System.IO;  
  
class Agenda  
{  
    private List<Contacto> contactos = new List<Contacto>();  
    private string rutaArchivo;  
  
    // El constructor recibe la ruta del archivo de datos  
    public Agenda(string rutaArchivo)  
    {  
        this.rutaArchivo = rutaArchivo;  
    }  
  
    public void Agregar(Contacto contacto) { contactos.Add(contacto); }  
    public int Cantidad { get { return contactos.Count; } }  
  
    public List<Contacto> ObtenerTodos()  
    {  
        return new List<Contacto>(contactos);  
    }  
  
    public List<Contacto> BuscarPorNombre(string texto)  
    {  
        List<Contacto> encontrados = new List<Contacto>();  
        foreach (Contacto c in contactos)  
        {  
            if (c.Nombre.IndexOf(texto, StringComparison.OrdinalIgnoreCase) >= 0)  
            {  
                encontrados.Add(c);  
            }  
        }  
    }  
}
```

```

        return encontrados;
    }

    public bool Eliminar(string nombre)
    {
        for (int i = 0; i < contactos.Count; i++)
        {
            if (contactos[i].Nombre.Equals(nombre,
StringComparison.OrdinalIgnoreCase))
            {
                contactos.RemoveAt(i);
                return true;
            }
        }
        return false;
    }

    // PERSISTENCIA: guardar toda la lista en el archivo (una línea por contacto)
    public void Guardar()
    {
        try
        {
            List<string> lineas = new List<string>();
            foreach (Contacto c in contactos)
            {
                lineas.Add(c.ALinea());
            }
            // WriteAllLines crea o reemplaza el archivo con todas las líneas
            File.WriteAllLines(rutaArchivo, lineas);
        }
        catch (Exception ex)
        {
            // Si no se puede guardar, avisamos: perder datos en silencio es grave
            Console.WriteLine("No se pudo guardar la agenda: " + ex.Message);
        }
    }

    // PERSISTENCIA: cargar los contactos desde el archivo al iniciar
    public void Cargar()
    {
        try
        {
            // Si el archivo no existe, es la primera vez: empezamos vacíos
            if (!File.Exists(rutaArchivo))
            {

```

```

        return;
    }
    string[] lineas = File.ReadAllLines(rutaArchivo);
    foreach (string linea in lineas)
    {
        // Ignoramos líneas vacías o corruptas en lugar de tumbar el programa
        if (string.IsNullOrEmpty(linea))
        {
            continue;
        }
        Contacto c = Contacto.DesdeLinea(linea);
        if (c != null)
        {
            contactos.Add(c);
        }
    }
    Console.WriteLine("Se cargaron " + contactos.Count + " contactos del
archivo.");
}
catch (Exception ex)
{
    Console.WriteLine("No se pudo cargar la agenda: " + ex.Message);
}
}
}

```

El diseño de persistencia es deliberadamente tolerante: líneas corruptas se ignoran, archivo inexistente significa agenda vacía (no error), y cualquier fallo de disco se reporta sin tumbar el programa. En una agenda real con miles de contactos usarías una base de datos, pero el patrón es el mismo: cargar al inicio, trabajar en memoria, guardar los cambios.

Etapas 4: el programa completo con menú. Unimos todo: la clase Contacto, la clase Agenda con persistencia y un Main con menú que orquesta. Este es el código final, comentado de principio a fin:

```

using System;
using System.Collections.Generic;
using System.IO;

class Contacto
{
    public string Nombre { get; set; }
    public string Telefono { get; set; }
    public string Correo { get; set; }
}

```

```

public Contacto(string nombre, string telefono, string correo)
{
    Nombre = nombre;
    Telefono = telefono;
    Correo = correo;
}

public override string ToString()
{
    return Nombre + " | Tel: " + Telefono + " | Correo: " + Correo;
}

public string ALinea()
{
    return Nombre + ";" + Telefono + ";" + Correo;
}

public static Contacto DesdeLinea(string linea)
{
    string[] partes = linea.Split(';');
    if (partes.Length != 3)
    {
        return null;
    }
    return new Contacto(partes[0], partes[1], partes[2]);
}
}

class Agenda
{
    private List<Contacto> contactos = new List<Contacto>();
    private string rutaArchivo;

    public Agenda(string rutaArchivo)
    {
        this.rutaArchivo = rutaArchivo;
    }

    public void Agregar(Contacto contacto) { contactos.Add(contacto); }
    public int Cantidad { get { return contactos.Count; } }
    public List<Contacto> ObtenerTodos() { return new List<Contacto>(contactos); }

    public List<Contacto> BuscarPorNombre(string texto)
    {
        List<Contacto> encontrados = new List<Contacto>();

```

```

        foreach (Contacto c in contactos)
        {
            if (c.Nombre.IndexOf(texto, StringComparison.OrdinalIgnoreCase) >= 0)
            {
                encontrados.Add(c);
            }
        }
        return encontrados;
    }

    public bool Eliminar(string nombre)
    {
        for (int i = 0; i < contactos.Count; i++)
        {
            if (contactos[i].Nombre.Equals(nombre,
StringComparison.OrdinalIgnoreCase))
            {
                contactos.RemoveAt(i);
                return true;
            }
        }
        return false;
    }

    public void Guardar()
    {
        try
        {
            List<string> lineas = new List<string>();
            foreach (Contacto c in contactos)
            {
                lineas.Add(c.ALinea());
            }
            File.WriteAllLines(rutaArchivo, lineas);
        }
        catch (Exception ex)
        {
            Console.WriteLine("No se pudo guardar la agenda: " + ex.Message);
        }
    }

    public void Cargar()
    {
        try
        {

```

```

        if (!File.Exists(rutaArchivo))
        {
            return;
        }
        foreach (string linea in File.ReadAllLines(rutaArchivo))
        {
            if (string.IsNullOrEmpty(linea))
            {
                continue;
            }
            Contacto c = Contacto.DesdeLinea(linea);
            if (c != null)
            {
                contactos.Add(c);
            }
        }
        Console.WriteLine("Se cargaron " + contactos.Count + " contactos.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("No se pudo cargar la agenda: " + ex.Message);
    }
}

class Programa
{
    // Lee un texto no vacío: no acepta campos en blanco
    static string LeerTexto(string mensaje)
    {
        while (true)
        {
            Console.Write(mensaje);
            string entrada = Console.ReadLine();
            if (!string.IsNullOrEmpty(entrada))
            {
                return entrada.Trim(); // Trim quita espacios de los extremos
            }
            Console.WriteLine("El campo no puede estar vacío.");
        }
    }

    // Valida que el campo no rompa el formato del archivo
    static string LeerCampo(string mensaje)
    {

```

```

while (true)
{
    string valor = LeerTexto(mensaje);
    if (valor.Contains(";"))
    {
        Console.WriteLine("El carácter ';' no está permitido.");
        continue;
    }
    return valor;
}

static void MostrarContactos(List<Contacto> lista)
{
    if (lista.Count == 0)
    {
        Console.WriteLine("(No hay contactos para mostrar.)");
        return;
    }
    int i = 1;
    foreach (Contacto c in lista)
    {
        Console.WriteLine(i + ". " + c);
        i++;
    }
}

static void Main()
{
    Console.WriteLine("=== AGENDA DE CONTACTOS ===");

    // La agenda vive en un archivo junto al programa
    Agenda agenda = new Agenda("agenda.txt");
    agenda.Cargar(); // Cargar al inicio: los datos persisten entre sesiones

    bool salir = false;
    while (!salir)
    {
        Console.WriteLine();
        Console.WriteLine("1. Agregar contacto");
        Console.WriteLine("2. Ver todos");
        Console.WriteLine("3. Buscar por nombre");
        Console.WriteLine("4. Eliminar contacto");
        Console.WriteLine("0. Guardar y salir");
        Console.Write("Elige una opción: ");
    }
}

```

```

string opcion = Console.ReadLine();

switch (opcion)
{
    case "1":
        // CREATE: pedimos los datos validados y agregamos
        string nombre = LeerCampo("Nombre: ");
        string telefono = LeerCampo("Teléfono: ");
        string correo = LeerCampo("Correo: ");
        agenda.Agregar(new Contacto(nombre, telefono, correo));
        agenda.Guardar(); // Guardamos tras cada cambio: nada se pierde
        Console.WriteLine("Contacto agregado y guardado.");
        break;

    case "2":
        // READ: listar todo
        Console.WriteLine("--- Todos los contactos (" +
            agenda.Cantidad + ") ---");
        MostrarContactos(agenda.ObtenerTodos());
        break;

    case "3":
        // READ: búsqueda parcial por nombre
        string texto = LeerTexto("Texto a buscar: ");
        List<Contacto> encontrados = agenda.BuscarPorNombre(texto);
        Console.WriteLine("--- Resultados (" + encontrados.Count + ")
---");

        MostrarContactos(encontrados);
        break;

    case "4":
        // DELETE: pedimos el nombre exacto y confirmamos
        string aBorrar = LeerTexto("Nombre exacto del contacto a eliminar:
");

        if (agenda.Eliminar(aBorrar))
        {
            agenda.Guardar();
            Console.WriteLine("Contacto eliminado y cambios guardados.");
        }
        else
        {
            Console.WriteLine("No se encontró ningún contacto con ese
nombre.");
        }
        break;
}

```



```

        case "0":
            // Guardar al salir como red de seguridad (ya guardamos en cada
cambio)

            agenda.Guardar();
            Console.WriteLine("Agenda guardada. ¡Hasta luego!");
            salir = true;
            break;

        default:
            Console.WriteLine("Opción no válida.");
            break;
    }
}
}
}
}

```

Este programa ya es software de verdad: persiste datos, valida entradas, maneja errores de archivo y está dividido en clases con responsabilidades claras. Si lo ejecutas, agregas contactos, lo cierras y lo vuelves a abrir, tus contactos siguen ahí. Esa sensación (cerrar el programa y que tus datos sobrevivan) es el momento en que un estudiante se convierte en programador: construiste algo que vive más allá de la ejecución.

Dos decisiones de diseño que vale la pena notar: guardamos después de cada cambio (agregar/eliminar) en lugar de solo al salir, porque si el programa se cierra de forma inesperada no se pierde nada; y separamos LeerTexto (no vacío) de LeerCampo (no vacío + sin punto y coma), componiendo validaciones en capas en lugar de repetir código. Pequeñas decisiones, gran diferencia en calidad.

Práctica

Estos ejercicios son extensiones de los dos proyectos: primero haces que funcionen, luego los mejoras. Es exactamente así como evoluciona el software real. Para cada uno, parte del código final de la unidad y agrega la funcionalidad pedida. Intenta cada uno antes de ver la solución.

1. Historial de la calculadora. Agrega a la calculadora una lista que guarde cada operación realizada como texto (por ejemplo "5 + 3 = 8"). Agrega la opción 7 "Ver historial" al menú, que muestre todas las operaciones realizadas en la sesión, y la opción 8 "Limpiar historial". El historial debe guardarse solo en memoria (se pierde al cerrar el programa). Pista: usa una `List<string>` y agrega la línea del resultado cada vez que una operación tiene éxito.

2. Resultado anterior (ANS). Modifica LeerNumero en la calculadora para que, si el usuario escribe la palabra "ans", se use el resultado de la operación anterior en lugar de pedir un número nuevo. Si aún no hay resultado anterior, muestra un mensaje y vuelve a pedir. Esto permite encadenar operaciones: calcular $5 + 3$, y luego escribir "ans * 2" para obtener 16. Pista: guarda el último resultado en una variable double? (anulable) a nivel de clase o pásala como parámetro.
3. Editar contactos (completar el CRUD). Nuestra agenda tiene Crear, Leer y Eliminar, pero le falta Actualizar. Agrega la opción 5 "Editar contacto": pide el nombre exacto del contacto, si existe muestra sus datos actuales y permite cambiar teléfono y correo (si el usuario deja el campo vacío, se conserva el valor anterior). Guarda los cambios en el archivo. Con esto la agenda implementa el CRUD completo.
4. Exportar la agenda a CSV. Agrega la opción 6 "Exportar a CSV": escribe todos los contactos en un archivo "agenda.csv" con una primera línea de encabezados "Nombre;Telefono;Correo" y luego un contacto por línea. El archivo debe poder abrirse en Excel. Además, maneja el caso de la agenda vacía con un mensaje en lugar de crear un archivo vacío. Este ejercicio te enseña el formato de intercambio de datos más usado en el mundo real.
5. Agenda ordenada y con conteo por letra. Modifica la opción "Ver todos" para que muestre los contactos ordenados alfabéticamente por nombre usando LINQ (OrderBy). Además, al final muestra un resumen: "Total: X contactos". Como reto extra, muestra cuántos contactos empiezan con cada letra inicial (puedes agrupar con GroupBy o con un ciclo). Pista: agenda.ObtenerTodos().OrderBy(c => c.Nombre).
6. Calculadora con memoria (M+, MR, MC). Agrega a la calculadora una "memoria" como las calculadoras físicas: una variable double memoria que inicia en 0. Agrega tres opciones al menú: "M+ Sumar a memoria" (suma el último resultado a la memoria), "MR Recuperar memoria" (muestra el valor guardado) y "MC Limpiar memoria" (la pone en 0). Si no hay último resultado cuando se presiona M+, muestra un mensaje. Este ejercicio practica el estado del programa más allá de una sola operación.
7. Reto integrador: mini inventario. Construye un programa nuevo aplicando todo lo aprendido: un inventario de productos para una tiendita. Cada producto tiene nombre, precio y cantidad en existencia. El menú debe permitir: agregar producto, vender (resta cantidad, sin permitir vender más de lo que hay), surtir (suma cantidad), ver inventario (con el valor total = precio * cantidad por producto y el valor total del inventario), y guardar/cargar en archivo automáticamente como la agenda. Si la cantidad llega a 0 al vender, muestra una alerta de "producto agotado". Este es tu primer programa "de negocio": si lo terminas, ya puedes decir que programas de verdad.

Soluciones

Solución 1: Historial de la calculadora. La idea es centralizar: cada vez que se calcula un resultado con éxito, se guarda la línea de texto en una `List<string>`. La opción 7 la muestra y la 8 la limpia.

```
using System;
using System.Collections.Generic;

class CalculadoraConHistorial
{
    // El historial vive mientras el programa corre (memoria, no archivo)
    static List<string> historial = new List<string>();
    static double? ultimoResultado = null;

    static double LeerNumero(string mensaje)
    {
        while (true)
        {
            Console.Write(mensaje);
            string entrada = Console.ReadLine();

            // Soporte para "ans": usa el resultado anterior
            if (entrada.Trim().ToLower() == "ans")
            {
                if (ultimoResultado.HasValue)
                {
                    Console.WriteLine("(usando " + ultimoResultado.Value + ")");
                    return ultimoResultado.Value;
                }
                Console.WriteLine("Aún no hay resultado anterior.");
                continue;
            }

            double numero;
            if (double.TryParse(entrada, out numero))
            {
                return numero;
            }
            Console.WriteLine("Eso no es un número válido. Intenta de nuevo.");
        }
    }

    // Guarda la operación en el historial y actualiza el último resultado
    static void Registrar(string operacion, double resultado)
```

```

{
    string linea = operacion + " = " + resultado;
    historial.Add(linea);
    ultimoResultado = resultado;
    Console.WriteLine(linea);
}

static void VerHistorial()
{
    Console.WriteLine("--- Historial (" + historial.Count + " operaciones) ---");
    if (historial.Count == 0)
    {
        Console.WriteLine("(Vacío: aún no haces operaciones.)");
        return;
    }
    int i = 1;
    foreach (string linea in historial)
    {
        Console.WriteLine(i + ". " + linea);
        i++;
    }
}

static void Main()
{
    while (true)
    {
        Console.WriteLine();
        Console.WriteLine("=== CALCULADORA CON HISTORIAL ===");
        Console.WriteLine("1. Sumar  2. Restar  3. Multiplicar  4. Dividir");
        Console.WriteLine("5. Ver historial  6. Limpiar historial  0. Salir");
        Console.Write("Elige: ");
        string opcion = Console.ReadLine();

        if (opcion == "0") break;

        if (opcion == "5") { VerHistorial(); continue; }
        if (opcion == "6")
        {
            historial.Clear();
            Console.WriteLine("Historial limpio.");
            continue;
        }

        if (opcion != "1" && opcion != "2" && opcion != "3" && opcion != "4")

```

```

    {
        Console.WriteLine("Opción no válida.");
        continue;
    }

    double a = LeerNumero("Primer número (o 'ans'): ");
    double b = LeerNumero("Segundo número (o 'ans'): ");

    switch (opcion)
    {
        case "1": Registrar(a + " + " + b, a + b); break;
        case "2": Registrar(a + " - " + b, a - b); break;
        case "3": Registrar(a + " * " + b, a * b); break;
        case "4":
            if (b == 0)
            {
                Console.WriteLine("Error: no se puede dividir entre cero.");
            }
            else
            {
                Registrar(a + " / " + b, a / b);
            }
            break;
    }
}
Console.WriteLine(";Hasta luego!");
}
}

```

Explicación: el método Registrar es el corazón de la solución: construye la línea "5 + 3 = 8", la agrega al historial, actualiza ultimoResultado y la muestra. Como todas las operaciones pasan por Registrar, ninguna se escapa del historial. Observa que esta solución también incluye el ejercicio 2 (el soporte de "ans" en LeerNumero usando double? anulable): en la práctica real, las mejoras se combinan. El historial en memoria es intencional: te enseña la diferencia entre estado de sesión (se pierde al cerrar) y persistencia (sobrevive en archivo, como la agenda).

Solución 2: Resultado anterior (ANS). Ya está integrada en la solución 1, pero veamos la pieza clave aislada para entenderla bien. El truco es double? (double anulable): puede guardar un número o valer null cuando aún no hay resultado.

```

using System;

class Programa
{
    // double? puede ser null: "aún no hay resultado"

```

```

static double? ultimoResultado = null;

static double LeerNumero(string mensaje)
{
    while (true)
    {
        Console.Write(mensaje);
        string entrada = Console.ReadLine().Trim();

        // Si escribe "ans", devolvemos el resultado anterior
        if (entrada.ToLower() == "ans")
        {
            if (ultimoResultado.HasValue)
            {
                return ultimoResultado.Value; // .Value extrae el double
            }
            Console.WriteLine("Aún no hay resultado anterior. Haz una operación
primero.");
            continue; // Vuelve a pedir
        }

        double numero;
        if (double.TryParse(entrada, out numero))
        {
            return numero;
        }
        Console.WriteLine("Eso no es un número válido.");
    }
}

static void Main()
{
    // Ejemplo de uso encadenado
    double a = LeerNumero("Primer número: "); // usuario: 5
    double b = LeerNumero("Segundo número: "); // usuario: 3
    ultimoResultado = a + b;
    Console.WriteLine("Resultado: " + ultimoResultado); // 8

    double c = LeerNumero("Otro número (prueba 'ans'): "); // usuario: ans -> 8
    Console.WriteLine("Usaste: " + c);
    ultimoResultado = ultimoResultado.Value * 2;
    Console.WriteLine("Doble del anterior: " + ultimoResultado); // 16
}
}

```

Explicación: HasValue dice si hay valor guardado y .Value lo extrae. Este patrón de "anulable como estado opcional" aparece en todo C#: configuraciones que pueden no existir, resultados de búsquedas, cachés. La lección general es que el estado del programa (lo que "recuerda" entre operaciones) vive en variables fuera de los métodos, y diseñar bien ese estado es gran parte de diseñar bien un programa.

Solución 3: Editar contactos. Necesitamos un método en Agenda que busque y devuelva el contacto (no solo bool), para poder modificarlo. Como Contacto es una clase (tipo por referencia), modificar el objeto devuelto modifica el de la lista.

```
// Método nuevo en la clase Agenda: busca y devuelve el contacto (o null)
public Contacto BuscarExacto(string nombre)
{
    foreach (Contacto c in contactos)
    {
        if (c.Nombre.Equals(nombre, StringComparison.OrdinalIgnoreCase))
        {
            return c; // Devuelve la referencia al objeto real de la lista
        }
    }
    return null;
}

// Opción 5 en el Main: editar contacto
case "5":
    string aEditar = LeerTexto("Nombre exacto del contacto a editar: ");
    Contacto contacto = agenda.BuscarExacto(aEditar);
    if (contacto == null)
    {
        Console.WriteLine("No se encontró ningún contacto con ese nombre.");
        break;
    }
    Console.WriteLine("Editando:" + contacto);
    Console.WriteLine("(Deja vacío para conservar el valor actual)");

    // Para teléfono y correo permitimos vacío = conservar
    Console.Write("Nuevo teléfono [" + contacto.Telefono + "]: ");
    string nuevoTel = Console.ReadLine();
    if (!string.IsNullOrEmpty(nuevoTel))
    {
        if (nuevoTel.Contains(";"))
        {
            Console.WriteLine("El ';' no está permitido. Se conserva el anterior.");
        }
        else
        {
            contacto.Telefono = nuevoTel;
        }
    }
    Console.WriteLine("Teléfono actualizado.");
}
```

```

        {
            contacto.Telefono = nuevoTel.Trim();
        }
    }

    Console.WriteLine("Nuevo correo [" + contacto.Correo + "]: ");
    string nuevoCorreo = Console.ReadLine();
    if (!string.IsNullOrEmpty(nuevoCorreo))
    {
        if (nuevoCorreo.Contains(";"))
        {
            Console.WriteLine("El ';' no está permitido. Se conserva el anterior.");
        }
        else
        {
            contacto.Correo = nuevoCorreo.Trim();
        }
    }

    agenda.Guardar();
    Console.WriteLine("Contacto actualizado: " + contacto);
    break;

```

Explicación: el punto clave es que `BuscarExacto` devuelve la referencia al objeto dentro de la lista (los `class` son tipos por referencia), así que al cambiar `contacto.Telefono` estamos cambiando el contacto guardado, sin necesidad de "reemplazarlo" en la lista. Después se guarda al archivo. El patrón "vacío = conservar" con el valor actual entre corchetes es el estándar en interfaces de consola y lo verás en herramientas profesionales. Con esta opción, la agenda ya es un CRUD completo: Create (agregar), Read (ver/buscar), Update (editar), Delete (eliminar).

Solución 4: Exportar a CSV. Reutilizamos `ALinea` y agregamos encabezados. La única diferencia con `agenda.txt` es la primera línea de títulos.


```
// Opción 6 en el Main: exportar a CSV
case "6":
    List<Contacto> todos = agenda.ObtenerTodos();
    if (todos.Count == 0)
    {
        Console.WriteLine("La agenda está vacía: no hay nada que exportar.");
        break;
    }
    try
    {
        List<string> lineas = new List<string>();
        // Encabezados: Excel los usa como títulos de columna
        lineas.Add("Nombre;Telefono;Correo");
        foreach (Contacto c in todos)
        {
            lineas.Add(c.ALinea());
        }
        File.WriteAllLines("agenda.csv", lineas);
        Console.WriteLine("Exportados " + todos.Count +
            " contactos a agenda.csv. Puedes abrirlo en Excel.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("No se pudo exportar: " + ex.Message);
    }
    break;
```

Explicación: el CSV es el "idioma común" de los datos tabulares: Excel, Google Sheets y casi cualquier sistema lo entienden. Nuestra agenda.txt ya era técnicamente un CSV con punto y coma; al agregarle encabezados se vuelve compatible con Excel en español (que espera punto y coma por la configuración regional). Este ejercicio parece pequeño, pero te enseña interoperabilidad: tus programas no viven aislados, y exportar datos en formatos estándar es una habilidad que piden en trabajos reales ("necesitamos sacar este reporte a Excel").

Solución 5: Agenda ordenada con resumen. Usamos LINQ para ordenar y agrupar. Solo cambiamos la opción 2; el resto queda igual.

```

using System.Linq; // Necesario para OrderBy y GroupBy

// Opción 2 mejorada: ver todos ordenados + resumen
case "2":
    List<Contacto> ordenados = agenda.ObtenerTodos()
                                   .OrderBy(c => c.Nombre)
                                   .ToList();
    Console.WriteLine("--- Contactos ordenados (" + ordenados.Count + ") ---");
    MostrarContactos(ordenados);

    if (ordenados.Count > 0)
    {
        Console.WriteLine();
        Console.WriteLine("Total: " + ordenados.Count + " contactos.");

        // Conteo por letra inicial: agrupamos por la primera letra
        var grupos = ordenados.GroupBy(c => char.ToUpper(c.Nombre[0]))
                               .OrderBy(g => g.Key);
        Console.WriteLine("Por letra inicial:");
        foreach (var grupo in grupos)
        {
            Console.WriteLine("  " + grupo.Key + ": " + grupo.Count());
        }
    }
    break;

```

Explicación: `OrderBy(c => c.Nombre)` ordena alfabéticamente con una lambda (mira cómo los delegados de la Unidad 9 reaparecen en proyectos reales). `GroupBy` agrupa por la primera letra convertida a mayúscula para que "ana" y "Ana" caigan en el mismo grupo. El resultado es un mini-reportaje, el tipo de funcionalidad que distingue a una aplicación útil de un simple guardadatos. Nota que no modificamos la lista interna de la agenda: `ObtenerTodos` devuelve una copia y el ordenamiento se hace sobre esa copia. La fuente de datos queda intacta.

Solución 6: Calculadora con memoria. Tres variables de estado (`memoria` y `ultimoResultado`) y tres opciones nuevas. Es un buen ejercicio de "estado del programa".

```

using System;

class CalculadoraConMemoria
{
    static double memoria = 0;
    static double? ultimoResultado = null;

    static void Main()

```

```

{
    while (true)
    {
        Console.WriteLine();
        Console.WriteLine("=== CALCULADORA (Memoria: " + memoria + ") ===");
        Console.WriteLine("1. Sumar  2. Restar  3. Multiplicar  4. Dividir");
        Console.WriteLine("5. M+ (sumar último resultado a memoria)");
        Console.WriteLine("6. MR (mostrar memoria)");
        Console.WriteLine("7. MC (limpiar memoria)");
        Console.WriteLine("0. Salir");
        Console.Write("Elige: ");
        string opcion = Console.ReadLine();

        if (opcion == "0") break;

        switch (opcion)
        {
            case "5":
                // M+: acumula el último resultado en la memoria
                if (ultimoResultado.HasValue)
                {
                    memoria += ultimoResultado.Value;
                    Console.WriteLine("Memoria ahora vale: " + memoria);
                }
                else
                {
                    Console.WriteLine("No hay resultado reciente para guardar.");
                }
                break;

            case "6":
                // MR: muestra la memoria
                Console.WriteLine("Memoria: " + memoria);
                break;

            case "7":
                // MC: limpia la memoria
                memoria = 0;
                Console.WriteLine("Memoria limpiada.");
                break;

            case "1":
            case "2":
            case "3":
            case "4":

```

```

        double a = LeerNumero("Primer número: ");
        double b = LeerNumero("Segundo número: ");
        double resultado = 0;
        bool ok = true;
        switch (opcion)
        {
            case "1": resultado = a + b; break;
            case "2": resultado = a - b; break;
            case "3": resultado = a * b; break;
            case "4":
                if (b == 0)
                {
                    Console.WriteLine("Error: no se puede dividir entre
cero.");
                    ok = false;
                }
                else resultado = a / b;
                break;
        }
        if (ok)
        {
            ultimoResultado = resultado;
            Console.WriteLine("Resultado: " + resultado);
        }
        break;

    default:
        Console.WriteLine("Opción no válida.");
        break;
    }
}

static double LeerNumero(string mensaje)
{
    while (true)
    {
        Console.Write(mensaje);
        double numero;
        if (double.TryParse(Console.ReadLine(), out numero))
        {
            return numero;
        }
        Console.WriteLine("Eso no es un número válido.");
    }
}

```

```
}  
}
```

Explicación: la memoria es estado que vive entre operaciones, igual que el historial del ejercicio 1. Mostrar su valor en el título del menú ("Memoria: 25") es un detalle de experiencia de usuario: el usuario siempre sabe qué hay guardado sin tener que preguntar. El patrón M+/MR/MC viene de las calculadoras físicas y es un ejemplo de cómo el software a menudo modela objetos del mundo real. Si combinas este ejercicio con el historial, tu calculadora ya tiene más funciones que muchas apps de calculadora básicas.

Solución 7: Reto integrador, mini inventario. Este es el más largo porque es un programa completo nuevo, pero notarás que el 80% son piezas que ya conoces: clase de modelo (como Contacto), clase gestora (como Agenda), persistencia en archivo y menú. Lo nuevo es la lógica de negocio (vender/surtir con validaciones).

```
using System;  
using System.Collections.Generic;  
using System.IO;  
  
// Modelo: un producto de la tiendita  
class Producto  
{  
    public string Nombre { get; set; }  
    public double Precio { get; set; }  
    public int Cantidad { get; set; }  
  
    public Producto(string nombre, double precio, int cantidad)  
    {  
        Nombre = nombre;  
        Precio = precio;  
        Cantidad = cantidad;  
    }  
  
    // Valor total de este producto en inventario  
    public double ValorTotal()  
    {  
        return Precio * Cantidad;  
    }  
  
    public override string ToString()  
    {  
        return Nombre + " | Precio: $" + Precio +  
            " | Existencia: " + Cantidad +  
            " | Valor: $" + ValorTotal().ToString("0.00");  
    }  
}
```

```

public string ALinea()
{
    return Nombre + ";" + Precio + ";" + Cantidad;
}

public static Producto DesdeLinea(string linea)
{
    string[] partes = linea.Split(';');
    if (partes.Length != 3)
    {
        return null;
    }
    double precio;
    int cantidad;
    // Si precio o cantidad vienen corruptos, la línea no sirve
    if (!double.TryParse(partes[1], out precio) ||
        !int.TryParse(partes[2], out cantidad))
    {
        return null;
    }
    return new Producto(partes[0], precio, cantidad);
}
}

class Inventario
{
    private List<Producto> productos = new List<Producto>();
    private string rutaArchivo;

    public Inventario(string rutaArchivo)
    {
        this.rutaArchivo = rutaArchivo;
    }

    public void Agregar(Producto p) { productos.Add(p); }

    public Producto Buscar(string nombre)
    {
        foreach (Producto p in productos)
        {
            if (p.Nombre.Equals(nombre, StringComparison.OrdinalIgnoreCase))
            {
                return p;
            }
        }
    }
}

```

```

    }
    return null;
}

public List<Producto> ObtenerTodos()
{
    return new List<Producto>(productos);
}

// Vender: resta cantidad; valida existencia suficiente
public bool Vender(string nombre, int cantidad)
{
    Producto p = Buscar(nombre);
    if (p == null)
    {
        Console.WriteLine("No existe ese producto.");
        return false;
    }
    if (cantidad <= 0)
    {
        Console.WriteLine("La cantidad debe ser mayor a cero.");
        return false;
    }
    if (p.Cantidad < cantidad)
    {
        Console.WriteLine("No hay suficiente existencia. Solo quedan " +
            p.Cantidad + ".");
        return false;
    }
    p.Cantidad -= cantidad;
    Console.WriteLine("Venta registrada: " + cantidad + " x " + p.Nombre +
        " = $" + (p.Precio * cantidad).ToString("0.00"));
    if (p.Cantidad == 0)
    {
        Console.WriteLine("¡ALERTA! " + p.Nombre + " se ha AGOTADO.");
    }
    else if (p.Cantidad <= 5)
    {
        Console.WriteLine("Aviso: quedan pocas unidades de " + p.Nombre +
            " (" + p.Cantidad + "). Considera surtir.");
    }
    return true;
}

// Surtir: suma cantidad al inventario

```

```

public bool Surtir(string nombre, int cantidad)
{
    Producto p = Buscar(nombre);
    if (p == null)
    {
        Console.WriteLine("No existe ese producto.");
        return false;
    }
    if (cantidad <= 0)
    {
        Console.WriteLine("La cantidad debe ser mayor a cero.");
        return false;
    }
    p.Cantidad += cantidad;
    Console.WriteLine("Inventario actualizado: " + p.Nombre +
        " ahora tiene " + p.Cantidad + " unidades.");
    return true;
}

// Valor total de todo el inventario
public double ValorInventario()
{
    double total = 0;
    foreach (Producto p in productos)
    {
        total += p.ValorTotal();
    }
    return total;
}

public void Guardar()
{
    try
    {
        List<string> lineas = new List<string>();
        foreach (Producto p in productos)
        {
            lineas.Add(p.ALinea());
        }
        File.WriteAllLines(rutaArchivo, lineas);
    }
    catch (Exception ex)
    {
        Console.WriteLine("No se pudo guardar: " + ex.Message);
    }
}

```



```

    }

    public void Cargar()
    {
        try
        {
            if (!File.Exists(rutaArchivo))
            {
                return;
            }
            foreach (string linea in File.ReadAllLines(rutaArchivo))
            {
                if (string.IsNullOrEmpty(linea))
                {
                    continue;
                }
                Producto p = Producto.DesdeLinea(linea);
                if (p != null)
                {
                    productos.Add(p);
                }
            }

            Console.WriteLine("Inventario cargado: " + productos.Count + "
productos.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("No se pudo cargar: " + ex.Message);
        }
    }
}

class Programa
{
    static string LeerTexto(string mensaje)
    {
        while (true)
        {
            Console.Write(mensaje);
            string entrada = Console.ReadLine();
            if (!string.IsNullOrEmpty(entrada))
            {
                return entrada.Trim();
            }
            Console.WriteLine("No puede estar vacío.");
        }
    }
}

```

```

    }
}

static double LeerPrecio(string mensaje)
{
    while (true)
    {
        Console.Write(mensaje);
        double precio;
        if (double.TryParse(Console.ReadLine(), out precio) && precio >= 0)
        {
            return precio;
        }
        Console.WriteLine("Precio inválido (debe ser un número mayor o igual a
0).");
    }
}

static int LeerCantidad(string mensaje)
{
    while (true)
    {
        Console.Write(mensaje);
        int cantidad;
        if (int.TryParse(Console.ReadLine(), out cantidad) && cantidad >= 0)
        {
            return cantidad;
        }
        Console.WriteLine("Cantidad inválida (debe ser un entero mayor o igual a
0).");
    }
}

static void Main()
{
    Console.WriteLine("=== INVENTARIO DE LA TIENDITA ===");
    Inventario inv = new Inventario("inventario.txt");
    inv.Cargar();

    while (true)
    {
        Console.WriteLine();
        Console.WriteLine("1. Agregar producto");
        Console.WriteLine("2. Vender");
        Console.WriteLine("3. Surtir");
    }
}

```

```

Console.WriteLine("4. Ver inventario");
Console.WriteLine("0. Guardar y salir");
Console.Write("Elige: ");
string opcion = Console.ReadLine();

if (opcion == "0")
{
    inv.Guardar();
    Console.WriteLine("Inventario guardado. ¡Hasta luego!");
    break;
}

switch (opcion)
{
    case "1":
        string nombre = LeerTexto("Nombre del producto: ");
        if (inv.Buscar(nombre) != null)
        {
            Console.WriteLine("Ese producto ya existe. Usa 'Surtir' para
agregar unidades.");
            break;
        }
        double precio = LeerPrecio("Precio: $");
        int cantidad = LeerCantidad("Cantidad inicial: ");
        inv.Agregar(new Producto(nombre, precio, cantidad));
        inv.Guardar();
        Console.WriteLine("Producto agregado.");
        break;

    case "2":
        string aVender = LeerTexto("Producto a vender: ");
        int cv = LeerCantidad("¿Cuántas unidades? ");
        if (inv.Vender(aVender, cv))
        {
            inv.Guardar();
        }
        break;

    case "3":
        string aSurtir = LeerTexto("Producto a surtir: ");
        int cs = LeerCantidad("¿Cuántas unidades llegan? ");
        if (inv.Surtir(aSurtir, cs))
        {
            inv.Guardar();
        }

```

```

        break;

    case "4":
        List<Producto> todos = inv.ObtenerTodos();
        Console.WriteLine("--- Inventario ---");
        if (todos.Count == 0)
        {
            Console.WriteLine("(Vacío: agrega productos primero.)");
        }
        foreach (Producto p in todos)
        {
            Console.WriteLine(p);
        }
        Console.WriteLine("VALOR TOTAL DEL INVENTARIO: $" +
            inv.ValorInventario().ToString("0.00"));

        break;

    default:
        Console.WriteLine("Opción no válida.");
        break;
    }
}
}
}

```

Explicación: si comparas este programa con la agenda, verás el mismo esqueleto: modelo con ALinea/DesdeLinea, gestor con lista privada y persistencia, Main con menú y validaciones. Lo nuevo es la lógica de negocio: Vender valida existencia y alerta al agotar, Surtir suma, y ValorInventario calcula el dinero total inmovilizado en mercancía. Esa es la diferencia entre un ejercicio y una aplicación: la aplicación implementa reglas del mundo real ("no puedes vender lo que no tienes"). Si entendiste este programa completo sin perderte, felicidades: ya lees y escribes software de nivel profesional básico. Agrégalo a tu portafolio de GitHub: un inventario funcional impresiona más que cien ejercicios sueltos.

Palabras finales: el curso termina, el programador empieza

Esta es la última sección de contenido del libro, así que hablemos de frente. Terminar un curso no te hace programador; programar te hace programador. Lo que acabas de leer te dio el mapa y las herramientas, pero el territorio lo recorres tú, un proyecto a la vez, un error a la vez. Todos los programadores que admiras pasaron por lo mismo: programas que no compilaban, errores que parecían imposibles, noches peleando con un bug que al final era un punto y coma. La diferencia entre los que llegan y los que se quedan en el camino no es el talento: es seguir al día siguiente.

Te propongo un plan concreto para los próximos 90 días. Primer mes: repasa las unidades que más te costaron y termina los ejercicios de práctica que dejaste pendientes, especialmente los de las unidades 9 y 10. Sube la calculadora, la agenda y el inventario a GitHub con buenos READMEs. Segundo mes: elige tu rama (web con ASP.NET Core, móvil con MAUI, juegos con Unity) y construye un proyecto guiado por un tutorial, pero modifícalo: agrégale funciones propias, cámbiale el tema, rómpelo y arréglalo. Tercer mes: construye algo tuyo desde cero, sin tutorial, aunque sea pequeño. Una app que resuelva un problema real tuyo o de alguien que conozcas: el control de gastos de tu casa, el catálogo de productos de un familiar, un juego sencillo para practicar. Ese proyecto propio, con sus decisiones y sus errores, es el que te va a enseñar más y el que vas a presumir en entrevistas.

Y cuando te estanques (porque te vas a estancar, todos nos estancamos), recuerda las tres salidas: lee el mensaje de error completo en lugar de entrar en pánico, divide el problema hasta que sea tan pequeño que sea obvio, y pide ayuda mostrando lo que intentaste. La comunidad de programadores en español es enorme y generosa: foros, Discord, Stack Overflow en español. Preguntar bien ("esto intenté, esto esperaba, esto obtuve") te abre puertas; preguntar "no me funciona, ayuda" las cierra.

Lucas EGA Academy seguirá aquí con más cursos y más proyectos. Pero lo más importante ya lo tienes: sabes leer código, sabes escribirlo, sabes organizarlo en clases y métodos, sabes persistir datos, sabes trabajar con asincronía y genéricos, y sabes terminar proyectos. Eso te pone por delante del 90% de la gente que dice "quiero aprender a programar" y nunca pasa del primer tutorial. Ahora ve y construye. Nos vemos en el siguiente curso.

Resumen de la unidad

- Los proyectos separan "saber los temas" de "saber programar": construye siempre, no solo estudies.
- Trabaja por etapas: versión mínima funcional primero, luego menú, luego validación, luego refactorización; prueba cada etapa antes de seguir.
- Cada método debe tener una sola responsabilidad: los métodos de cálculo no saben de consola, la validación vive en un solo lugar, el Main solo orquesta.
- Valida todas las entradas del usuario con TryParse y ciclos: un usuario que escribe mal es lo normal, no un caso excepcional.
- Usa excepciones para condiciones excepcionales (división entre cero, raíz de negativo) y atrápalas en un punto central para que el programa nunca se caiga sin explicar.

- La refactorización no cambia lo que hace el programa: cambia lo fácil que es leerlo, probarlo y extenderlo; un Main de 12 líneas legibles vale más que uno de 60 mezcladas.
- CRUD (Crear, Leer, Actualizar, Eliminar) es el patrón detrás de casi todas las aplicaciones del mundo; la agenda lo implementa completo.
- La persistencia en archivos sigue el ciclo: cargar al iniciar, trabajar en memoria, guardar tras cada cambio; tolera archivos inexistentes y líneas corruptas sin tumbarse.
- El CSV con encabezados es el formato de intercambio de datos más universal: tus programas deben poder exportar a formatos que otros sistemas entiendan.
- El estado del programa (historial, memoria, último resultado) vive en variables fuera de los métodos; diseñarlo bien es gran parte de diseñar bien el software.
- Después del curso, el ecosistema .NET te espera: ASP.NET Core (APIs y backend), .NET MAUI (apps móviles y escritorio), Unity (videojuegos) y Blazor (web con C#).
- Tu portafolio en GitHub con proyectos terminados vale más que certificados: sube la calculadora, la agenda y el inventario con un buen README.
- Aprende Git, practica explicar tu código en voz alta y aplica a puestos junior sin miedo: la constancia construyendo proyectos es lo que consigue trabajo.

EXAMEN FINAL

Llegaste al final del libro. Este examen tiene 50 preguntas que cubren las 10 unidades: opción múltiple, verdadero o falso (con justificación) y preguntas abiertas donde tendrás que escribir o analizar código C#. Hazlo con calma y sin ver las respuestas: la gracia es descubrir qué dominas y qué necesitas repasar.

Instrucciones:

- Responde las 50 preguntas en orden, sin consultar el libro ni las soluciones.
- Tiempo recomendado: 2 horas. Si te atorras en una pregunta más de 5 minutos, márcala y sigue adelante.
- En las preguntas de verdadero o falso, escribe también una justificación corta: no basta con adivinar.
- En las preguntas abiertas de código, escribe el código completo como si fuera a compilarse, no solo la idea.
- Al terminar, revisa la sección "## Respuestas y soluciones" y cuenta tu puntaje: cada pregunta vale 2 puntos, total 100. 90 o más es excelente, 70 a 88 es bueno, 50 a 68 significa que debes repasar un par de unidades, y menos de 50 te recomiendo releer el libro completo antes de seguir.

Preguntas

PREGUNTA 1 (Opción múltiple): ¿Qué es .NET?

a) Un editor de código creado por Microsoft b) Una plataforma gratuita y de código abierto para crear aplicaciones con C# c) Un sistema operativo d) Un lenguaje de programación anterior a C#

PREGUNTA 2 (Verdadero/Falso): C# es un lenguaje compilado: tu código se traduce a un lenguaje intermedio (IL) que después ejecuta el CLR (Common Language Runtime). Justifica tu respuesta.

PREGUNTA 3 (Abierta): Escribe el programa mínimo en C# (usando top-level statements, sin declarar la clase Program) que imprima el texto "Hola, C#" en la consola.

PREGUNTA 4 (Opción múltiple): ¿Qué comando del SDK de .NET crea un nuevo proyecto de aplicación de consola?

a) dotnet new console b) dotnet create app c) dotnet console new d) dotnet build console

PREGUNTA 5 (Opción múltiple): Ya tienes tu proyecto creado y quieres compilarlo y ejecutarlo. ¿Qué comando usas?

a) dotnet run b) dotnet start c) dotnet exec d) dotnet go

PREGUNTA 6 (Opción múltiple): ¿Cuál de las siguientes declaraciones de variables es correcta en C#?

a) int edad = "25"; b) string nombre = "Ana"; c) bool activo = "true"; d) double pi = '3.14';

PREGUNTA 7 (Abierta): ¿Qué imprime el siguiente programa? Explica por qué.

```
int a = 7;
int b = 2;
Console.WriteLine(a / b);
Console.WriteLine(a % b);
```

PREGUNTA 8 (Verdadero/Falso): Si declaras una variable con "var", su tipo se decide cuando el programa corre y puede cambiar después a otro tipo. Justifica tu respuesta.

PREGUNTA 9 (Opción múltiple): ¿Qué hace la expresión `Convert.ToInt32(Console.ReadLine())`?

a) Convierte el texto que escribe el usuario en un número entero b) Convierte un número entero en texto c) Lee un número decimal del teclado d) Imprime un número entero en la consola

PREGUNTA 10 (Abierta): Escribe un programa completo que pida al usuario dos números enteros, los lea del teclado y muestre en la consola la suma de ambos con un mensaje claro.

PREGUNTA 11 (Opción múltiple): ¿Qué imprime el siguiente código?

a) A b) B c) Nada, porque hay un error de sintaxis d) Ambas letras

```
int x = 10;
if (x > 5 && x < 15)
    Console.WriteLine("A");
else
    Console.WriteLine("B");
```


PREGUNTA 12 (Verdadero/Falso): En un switch de C#, cada case necesita terminar con break (o con return, throw o goto) porque el lenguaje no permite que la ejecución "caiga" automáticamente al siguiente case. Justifica tu respuesta.

PREGUNTA 13 (Abierta): Escribe un ciclo while que imprima en la consola los números pares del 2 al 20, uno por línea.

PREGUNTA 14 (Opción múltiple): ¿Cuál es la diferencia entre while y do-while?

a) No hay ninguna diferencia, son sinónimos b) do-while ejecuta el cuerpo del ciclo al menos una vez, aunque la condición sea falsa desde el inicio c) while ejecuta el cuerpo al menos una vez, aunque la condición sea falsa desde el inicio d) do-while no permite usar una condición

PREGUNTA 15 (Abierta): ¿Qué imprime el siguiente programa? Sigue la ejecución paso a paso en tu cabeza antes de responder.

```
int i = 1;
int suma = 0;
while (i <= 4)
{
    suma = suma + i;
    i++;
}
Console.WriteLine(suma);
```

PREGUNTA 16 (Opción múltiple): ¿Cuántas veces se ejecuta el cuerpo del ciclo en el siguiente código?

a) 4 b) 5 c) 6 d) Infinitas veces

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

PREGUNTA 17 (Verdadero/Falso): Con foreach puedes modificar los elementos de la colección mientras la recorres, porque la variable de iteración es una referencia directa modificable. Justifica tu respuesta.

PREGUNTA 18 (Abierta): Escribe un método llamado EsPar que reciba un número entero como parámetro y devuelva true si el número es par y false si es impar. Incluye también una línea que lo pruebe con el número 8.

PREGUNTA 19 (Opción múltiple): ¿Qué imprime el siguiente programa?

a) 5 b) 8 c) 10 d) 12

```
static int Doble(int n)
{
    return n * 2;
}

Console.WriteLine(Doble(4) + Doble(1));
```

PREGUNTA 20 (Abierta): Explica con tus palabras la diferencia entre un parámetro y un valor de retorno en un método. Incluye un ejemplo corto de un método que use ambos.

PREGUNTA 21 (Opción múltiple): ¿Cuál es la forma correcta de declarar un arreglo de 5 números enteros en C#?

a) int[5] numeros; b) int[] numeros = new int[5]; c) array<int> numeros(5); d) int numeros = new int[5];

PREGUNTA 22 (Abierta): ¿Qué imprime el siguiente programa? Explica qué significa cada parte.

```
int[] n = { 2, 4, 6 };
Console.WriteLine(n[1] + n.Length);
```

PREGUNTA 23 (Verdadero/Falso): En C#, los índices de un arreglo empiezan en 1, así que el primer elemento se accede con arreglo[1]. Justifica tu respuesta.

PREGUNTA 24 (Abierta): Crea una clase llamada Perro con un atributo nombre de tipo string, un constructor que reciba el nombre y lo guarde, y un método Ladrar que imprima en la consola el texto "Guau, soy " seguido del nombre del perro. Incluye también el código que crea un objeto Perro llamado "Firulais" y llama a Ladrar.

PREGUNTA 25 (Opción múltiple): ¿Qué es un constructor en programación orientada a objetos?

a) Un método especial que destruye el objeto cuando ya no se usa b) Un método especial que se ejecuta automáticamente al crear un objeto y sirve para inicializar sus datos c) Una clase que no tiene atributos d) Un tipo especial de ciclo

PREGUNTA 26 (Opción múltiple): ¿Qué modificador de acceso hace que un miembro de una clase solo sea visible dentro de esa misma clase?

a) public b) private c) protected d) internal

PREGUNTA 27 (Verdadero/Falso): Una propiedad en C# permite controlar cómo se lee y cómo se escribe un dato, usando los bloques get y set. Justifica tu respuesta con un ejemplo breve.

PREGUNTA 28 (Abierta): Escribe una clase llamada Rectangulo con dos propiedades, Ancho y Alto, de tipo double. El set de cada propiedad no debe aceptar valores negativos: si alguien intenta asignar un negativo, el valor debe quedar en 0. Además, la clase debe tener un método CalcularArea que devuelva el área del rectángulo.

PREGUNTA 29 (Opción múltiple): ¿Qué imprime el siguiente programa?

a) Hola A b) Hola B c) Hola A seguido de Hola B d) Hay un error de compilación

```
class A
{
    public virtual void Saludar()
    {
        Console.WriteLine("Hola A");
    }
}

class B : A
{
    public override void Saludar()
    {
        Console.WriteLine("Hola B");
    }
}

A obj = new B();
obj.Saludar();
```

PREGUNTA 30 (Abierta): Explica con tus palabras qué es el polimorfismo en programación orientada a objetos. Da un ejemplo con una clase base y dos clases que hereden de ella, donde cada una se comporte de forma distinta al llamar al mismo método.

PREGUNTA 31 (Opción múltiple): ¿Qué es una interfaz en C#?

a) Una clase que solo tiene métodos estáticos b) Un contrato que define métodos y propiedades que una clase debe implementar, sin darles implementación c) Una ventana gráfica de la aplicación d) Un tipo especial de arreglo

PREGUNTA 32 (Abierta): ¿Qué imprime el siguiente programa? Explica cada línea.

```
List<int> numeros = new List<int> { 5, 10, 15 };
numeros.Add(20);
Console.WriteLine(numeros.Count);
```

PREGUNTA 33 (Verdadero/Falso): Con LINQ puedes filtrar una colección usando el método Where, sin necesidad de escribir manualmente un ciclo for o foreach para revisar elemento por elemento. Justifica tu respuesta.

PREGUNTA 34 (Abierta): Dada una lista de enteros llamada numeros, escribe una consulta LINQ que devuelva únicamente los números mayores que 10, ordenados de menor a mayor, en una nueva lista.

PREGUNTA 35 (Opción múltiple): ¿Qué hace el siguiente código?

a) Suma todos los números pares de la lista b) Crea una nueva lista que contiene solo los números pares c) Elimina los números pares de la lista original d) Ordena la lista de menor a mayor

```
List<int> numeros = new List<int> { 1, 2, 3, 4, 5, 6 };
var pares = numeros.Where(n => n % 2 == 0).ToList();
```

PREGUNTA 36 (Opción múltiple): En un bloque try-catch-finally, ¿qué bloque se ejecuta SIEMPRE, haya ocurrido una excepción o no?

a) try b) catch c) finally d) throw

PREGUNTA 37 (Abierta): ¿Qué imprime el siguiente programa? Explica por qué se ejecuta esa línea y no otra.

```
try
{
    int x = int.Parse("abc");
    Console.WriteLine("Conversión exitosa");
}
catch (FormatException)
{
    Console.WriteLine("Error de formato");
}
```

PREGUNTA 38 (Verdadero/Falso): El método File.WriteAllText sobrescribe el contenido del archivo si el archivo ya existe, y lo crea si no existe. Justifica tu respuesta.

PREGUNTA 39 (Abierta): Escribe un programa que lea todo el texto del archivo "datos.txt" y lo muestre en la consola. Si el archivo no existe, el programa debe mostrar el mensaje "El archivo no existe" en lugar de romperse. Usa manejo de excepciones.

PREGUNTA 40 (Opción múltiple): ¿Qué devuelve la expresión "hola mundo".ToUpper()?

a) "hola mundo" b) "HOLA MUNDO" c) "Hola Mundo" d) Produce un error de compilación

PREGUNTA 41 (Opción múltiple): ¿Qué significa marcar un método con la palabra `async`?

a) Que el método está escrito en otro lenguaje de programación b) Que el método puede usar `await` en su interior y no bloquea el hilo mientras espera una operación c) Que el método siempre corre más rápido que uno normal d) Que el método no puede devolver ningún valor

PREGUNTA 42 (Verdadero/Falso): Cuando el programa llega a un `await`, todo el programa se congela y nada más puede ejecutarse hasta que la tarea termina. Justifica tu respuesta.

PREGUNTA 43 (Abierta): Escribe un método asíncrono llamado `DescargarDatos` que simule una descarga esperando 2 segundos con `Task.Delay(2000)` y después devuelva el texto "descarga lista". El método debe poder usarse con `await`.

PREGUNTA 44 (Opción múltiple): ¿Qué es un delegado en C#?

a) Una variable que guarda una referencia a un método para poder llamarlo después b) Un tipo de clase abstracta c) Un ciclo especial para recorrer colecciones d) Un tipo de excepción

PREGUNTA 45 (Abierta): Explica con tus palabras qué es una clase genérica, usando `List<T>` como ejemplo. ¿Por qué es mejor usar `List<int>` que una lista de `object` para guardar números enteros? Menciona al menos dos razones.

PREGUNTA 46 (Opción múltiple): ¿Cuál de las siguientes es una buena práctica al organizar un proyecto de consola en C#?

a) Escribir todo el código en un solo archivo gigante para no perderse b) Separar las responsabilidades en clases con nombres claros, cada una en su propio archivo c) Usar nombres de variables de una sola letra para escribir más rápido d) Evitar crear métodos para que el programa sea más corto

PREGUNTA 47 (Abierta): Vas a crear un sistema de inventario que guarda productos en un archivo de texto. Describe qué clases crearías, qué datos guardaría cada una y qué métodos tendría. No necesitas escribir todo el código, pero sí la estructura completa con nombres de clases, atributos y métodos.

PREGUNTA 48 (Verdadero/Falso): Un programa bien diseñado debe validar la entrada del usuario antes de procesarla, en lugar de suponer que el usuario siempre escribirá datos correctos. Justifica tu respuesta con un ejemplo concreto.

PREGUNTA 49 (Abierta): ¿Qué imprime el siguiente programa completo? Sigue la ejecución línea por línea.

```
class Estudiante
{
    public string Nombre { get; set; }
    public int Calificacion { get; set; }

    public bool Aprobo()
    {
        return Calificacion >= 6;
    }
}

List<Estudiante> grupo = new List<Estudiante>
{
    new Estudiante { Nombre = "Ana", Calificacion = 9 },
    new Estudiante { Nombre = "Luis", Calificacion = 4 },
    new Estudiante { Nombre = "Marta", Calificacion = 7 }
};

foreach (var e in grupo)
{
    if (e.Aprobo())
        Console.WriteLine(e.Nombre + " aprobó");
    else
        Console.WriteLine(e.Nombre + " reprobó");
}
```

PREGUNTA 50 (Abierta): Diseña un menú de consola para una aplicación con 4 opciones: 1) Agregar producto, 2) Listar productos, 3) Buscar producto, 4) Salir. El menú debe repetirse hasta que el usuario elija salir, usando un ciclo do-while y un switch para atender cada opción. Escribe el código completo del menú (los métodos Agregar, Listar y Buscar pueden quedar como métodos vacíos con un mensaje).

Respuestas y soluciones

RESPUESTA 1: b) .NET es una plataforma gratuita y de código abierto para crear aplicaciones con C#. No es un editor de código (ese sería Visual Studio Code o Visual Studio), no es un sistema operativo y no es un lenguaje: es el conjunto de herramientas, bibliotecas y el entorno de ejecución (CLR) sobre el que corren los programas escritos en C#. Con .NET puedes crear aplicaciones de consola, web, escritorio, móviles y videojuegos, y el mismo código puede correr en Windows, Linux y macOS.

Dato útil para tu camino: .NET publica una versión nueva cada año en noviembre, y las versiones LTS (soporte a largo plazo) son las recomendadas para aprender y trabajar, porque reciben actualizaciones por tres años. Cuando instalas el SDK con el instalador oficial, obtienes el compilador, las bibliotecas y la herramienta dotnet en un solo paquete. No necesitas nada más para seguir este libro: todo lo que usamos funciona con la versión LTS más reciente.

RESPUESTA 2: Verdadero. C# es un lenguaje compilado en dos pasos: primero el compilador traduce tu código fuente a un lenguaje intermedio llamado IL (Intermediate Language), y cuando ejecutas el programa, el CLR (Common Language Runtime) traduce ese IL a código máquina justo a tiempo. Por eso necesitas tener instalado el runtime de .NET (o publicarlo incluido) para correr un programa de C# en una computadora.

Una ventaja poco conocida de este diseño en dos pasos es que varios lenguajes comparten el mismo runtime: C#, F# y Visual Basic se compilan al mismo IL y pueden usar las mismas bibliotecas. Por eso, cuando aprendes C# en realidad estás aprendiendo a usar todo el ecosistema .NET. Además, el CLR se encarga de tareas como la recolección de basura (liberar memoria que ya no usas), así que no tienes que administrar la memoria manualmente como en C o C++.

RESPUESTA 3:

```
Console.WriteLine("Hola, C#");
```

Esa sola línea es un programa completo válido en C# moderno gracias a los top-level statements: ya no es obligatorio escribir la clase Program ni el método Main para programas sencillos. El compilador genera ese código por ti detrás de escena. Si lo guardas en un archivo Program.cs dentro de un proyecto de consola y ejecutas "dotnet run", verás el mensaje en la terminal.

En proyectos modernos de .NET, el archivo Program.cs también incluye los 'usings implícitos': el compilador agrega automáticamente los espacios de nombres más comunes (como System), así que no necesitas escribir 'using System;' para usar Console. Si tu proyecto es más grande y prefieres la estructura clásica, puedes escribir la clase Program con el método Main manualmente; ambas formas son válidas y el resultado es el mismo. Para este libro usaremos top-level statements por simplicidad.

RESPUESTA 4: a) dotnet new console. Este comando crea la carpeta y los archivos base de un proyecto de aplicación de consola (el archivo .csproj y un Program.cs de ejemplo). Las otras opciones no existen: "dotnet create", "dotnet console new" y "dotnet build console" no son comandos válidos del SDK. Recuerda que "dotnet build" sí existe, pero solo compila un proyecto que ya existe, no crea uno nuevo.

El comando 'dotnet new' tiene una lista larga de plantillas: prueba 'dotnet new list' para verlas todas. Además de console existen classlib (biblioteca de clases), web (aplicación web vacía), mvc y otras. Cada plantilla genera la estructura mínima de archivos para ese tipo de proyecto. Cuando creas el proyecto, se genera un archivo .csproj que describe tu proyecto en formato XML: ahí se indica la versión de .NET destino y otras configuraciones. No necesitas editarlo a mano mientras aprendes, pero es bueno saber que existe.

RESPUESTA 5: a) dotnet run. Este comando compila tu proyecto y, si no hay errores, lo ejecuta de inmediato mostrando la salida en la terminal. Es el comando que más vas a usar mientras aprendes. "dotnet build" solo compila sin ejecutar, y "dotnet start", "dotnet exec" y "dotnet go" no existen como comandos del SDK.

Si tu código tiene errores de compilación, 'dotnet run' se detiene y te muestra los errores en rojo sin ejecutar nada: primero debes corregirlos. Para compilar sin ejecutar usa 'dotnet build', que es útil cuando solo quieres verificar que todo compila. Y si quieres el programa como un archivo ejecutable independiente para compartirlo, el comando es 'dotnet publish'. Mientras aprendes, quédate con dos: 'dotnet new console' para crear y 'dotnet run' para probar.

RESPUESTA 6: b) string nombre = "Ana";. Es la única declaración correcta. La opción a) falla porque no puedes guardar texto entre comillas en una variable int: el compilador exige que el valor coincida con el tipo. La opción c) falla por lo mismo: un bool solo acepta true o false sin comillas, no el texto "true". La opción d) falla porque las comillas simples definen un char (un solo carácter), no un double: un double se escribe como 3.14 sin comillas. Esta pregunta prueba la idea central del tipado fuerte de C#: cada tipo solo acepta valores de su tipo.

El error más frecuente aquí es confundir las comillas: en C# las comillas DOBLES (" ") son para string (texto de cualquier longitud) y las comillas SIMPLES (' ') son para char (un solo

carácter). Por eso '3.14' con comillas simples es un error: son cinco caracteres y un char solo admite uno. Otro detalle: los números decimales como 3.14 son double por defecto; si necesitas un float debes escribir 3.14f, y para dinero de alta precisión se usa decimal con el sufijo m (19.99m).

RESPUESTA 7: El programa imprime:

```
3
1
```

La primera línea imprime 3 porque al dividir dos enteros (int / int), C# hace división entera: trunca los decimales y se queda solo con la parte entera. 7 entre 2 es 3.5, pero como ambos son int, el resultado es 3. La segunda línea imprime 1 porque el operador % devuelve el residuo de la división: 7 entre 2 es 3 con residuo 1. Este es uno de los errores más comunes de principiantes: esperar 3.5 de una división entre enteros. Si quieres decimales, al menos uno de los operandos debe ser double.

Si necesitas el resultado con decimales, convierte al menos un operando a double antes de dividir: '(double)a / b' produce 3.5. El paréntesis con el tipo antes del valor se llama 'cast' o conversión explícita, y le dice al compilador 'trata este int como double por un momento'. Como b también se convierte automáticamente a double en la operación, la división se hace en decimales. Este truco aparece en todo el libro, por ejemplo al calcular promedios.

RESPUESTA 8: Falso. La palabra var no hace que el tipo se decida cuando el programa corre: el compilador infiere el tipo en tiempo de compilación a partir del valor inicial, y a partir de ese momento la variable tiene ese tipo fijo para siempre. Por ejemplo, con "var x = 5;" el compilador decide que x es int, y si después escribes "x = "hola";" obtendrás un error de compilación porque no puedes guardar texto en un int. var es solo un atajo de escritura, no una variable sin tipo.

Otro detalle importante: con var estás obligado a inicializar la variable en la misma línea ('var x;' sola es un error), porque sin un valor inicial el compilador no puede inferir el tipo. Y aunque var ahorra escritura, no abuses de él: en código profesional se recomienda usar var cuando el tipo es obvio ('var lista = new List<int>();') pero escribir el tipo explícito cuando aclara la lectura. La legibilidad siempre le gana al ahorro de tecleo.

RESPUESTA 9: a) Convierte el texto que escribe el usuario en un número entero. `Console.ReadLine()` siempre devuelve un string (texto), aunque el usuario escriba dígitos. `Convert.ToInt32` toma ese texto y lo transforma en un int para que puedas hacer operaciones matemáticas con él. Ojo: si el usuario escribe algo que no es un número, esta conversión lanza una excepción `FormatException` y el programa se rompe; por eso en la Unidad 8 aprendiste a validar la entrada.

Existe una alternativa más segura que verás en la Unidad 8: `int.TryParse`. En lugar de lanzar una excepción cuando el texto no es un número, `TryParse` devuelve true o false indicando si la conversión tuvo éxito, y entrega el resultado por un parámetro de salida. Es la forma profesional de leer números del usuario, porque los usuarios reales escriben letras, dejan vacío o ponen espacios. Regla de oro: si el dato viene de un humano, válidalo.

RESPUESTA 10:

```
Console.Write("Escribe el primer número: ");
int numero1 = Convert.ToInt32(Console.ReadLine());

Console.Write("Escribe el segundo número: ");
int numero2 = Convert.ToInt32(Console.ReadLine());

int suma = numero1 + numero2;
Console.WriteLine("La suma es: " + suma);
```

El programa usa `Console.Write` para mostrar el mensaje sin salto de línea, lee cada número con `Console.ReadLine` y lo convierte a int con `Convert.ToInt32`, los suma en una tercera variable y muestra el resultado. Fíjate en que sin la conversión no podrías sumar: estarías "sumando" textos, y el operador + entre strings los concatena en lugar de sumarlos.

Puedes escribir el mensaje final de forma más elegante con interpolación de cadenas: `Console.WriteLine($"La suma es: {suma}");`. El signo \$ antes de las comillas activa la interpolación y lo que está entre llaves se reemplaza por el valor de la variable. Es la forma moderna y preferida en C# porque evita concatenar con + y el código se lee mejor. La verás en casi todos los ejemplos del libro a partir de la Unidad 2.

RESPUESTA 11: a) A. La condición del if combina dos comparaciones con el operador **&&** (Y lógico): `x > 5` es verdadero (10 es mayor que 5) y `x < 15` también es verdadero (10 es menor que 15). Como ambas son verdaderas, la condición completa es verdadera y se ejecuta la primera rama, que imprime "A". El else solo se ejecutaría si al menos una de las dos comparaciones fuera falsa.

C# evalúa && con 'cortocircuito': si la primera condición es falsa, ni siquiera evalúa la segunda, porque el resultado ya no puede ser verdadero. Esto es útil para escribir condiciones seguras como 'if (texto != null && texto.Length > 0)': si texto es null, la segunda parte no se evalúa y evitas un error. Con || pasa lo mismo al revés: si la primera es verdadera, la segunda se omite. Aprovechalo para poner primero la condición más barata o la que protege a la otra.

RESPUESTA 12: Verdadero. A diferencia de C o C++, en C# no existe el "fall-through" implícito: el compilador te obliga a terminar cada case con break, return, throw o goto, y te marca error si se te olvida. Esto es una decisión de diseño para evitar uno de los errores más comunes y difíciles de detectar en otros lenguajes, donde olvidar un break hace que se ejecuten varios cases sin querer. Si de verdad quieres que dos cases hagan lo mismo, en C# los apilas uno tras otro sin código entre ellos.

En C# moderno también existe la 'expresión switch', una versión compacta que devuelve un valor directamente y que verás en código profesional. Pero el switch clásico con case y break sigue siendo el que debes dominar primero, porque deja clarísima la lógica de decisión múltiple. Consejo de examen: si ves un switch sin break en un case, no asumas que 'cae' al siguiente como en C: en C# eso es error de compilación, salvo que el case esté vacío y comparta el código con el siguiente.

RESPUESTA 13:

```
int numero = 2;
while (numero <= 20)
{
    Console.WriteLine(numero);
    numero = numero + 2;
}
```

El ciclo empieza en 2 y mientras el número sea menor o igual a 20 lo imprime y le suma 2, así solo aparecen los pares: 2, 4, 6... hasta 20. La línea "numero = numero + 2" es la más importante: sin ella el ciclo sería infinito porque la condición nunca cambiaría. Siempre verifica que tu while tenga algo que modifique la condición dentro del cuerpo.

El peligro número uno del while es el ciclo infinito: si olvidas la línea que modifica la variable de control, la condición nunca se vuelve falsa y el programa se queda girando para siempre (tendrás que detenerlo con Ctrl+C en la terminal). Cuando eso te pase, no te frustres: revísalo como un detective y pregunta '¿qué cambia en cada vuelta para que esto termine?'. Si la respuesta es 'nada', ahí está tu error. Todo ciclo necesita una condición de salida que avance.

RESPUESTA 14: b) do-while ejecuta el cuerpo del ciclo al menos una vez, aunque la condición sea falsa desde el inicio. La diferencia está en dónde se evalúa la condición: en while se revisa ANTES de entrar al cuerpo (si es falsa desde el principio, el cuerpo nunca se ejecuta), mientras que en do-while la condición se revisa DESPUÉS de ejecutar el cuerpo (así que la primera pasada está garantizada). Por eso do-while es ideal para menús: quieres mostrar el menú al menos una vez antes de preguntar si el usuario quiere salir.

Regla práctica para elegir: usa while cuando no sabes cuántas vueltas necesitarás (leer datos hasta que el usuario escriba 'salir', esperar una respuesta válida), usa do-while cuando el cuerpo debe ejecutarse al menos una vez sí o sí (mostrar un menú, pedir una contraseña), y usa for cuando sabes el número exacto de repeticiones (recorrer del 1 al 10). Elegir el ciclo adecuado no es solo estética: comunica tu intención a quien lea tu código.

RESPUESTA 15: El programa imprime 10. Sigamos la ejecución: i empieza en 1 y suma en 0. Primera vuelta: la condición $1 \leq 4$ es verdadera, $\text{suma} = 0 + 1 = 1$, i pasa a 2. Segunda vuelta: $2 \leq 4$ es verdadera, $\text{suma} = 1 + 2 = 3$, i pasa a 3. Tercera vuelta: $3 \leq 4$ es verdadera, $\text{suma} = 3 + 3 = 6$, i pasa a 4. Cuarta vuelta: $4 \leq 4$ es verdadera, $\text{suma} = 6 + 4 = 10$, i pasa a 5. Quinta revisión: $5 \leq 4$ es falsa, el ciclo termina y se imprime 10. Es la suma de $1 + 2 + 3 + 4$.

Este programa usa el patrón 'acumulador': una variable (suma) que empieza en un valor neutro (0) y va juntando resultados vuelta tras vuelta. Es uno de los patrones más usados en programación: contar elementos, sumar totales, concatenar textos. La otra variable (i) es el 'contador' que controla las vueltas. Cuando veas un while con estas dos variables, casi seguro estás ante un acumulador. Prácticalo modificando el programa para que multiplique en lugar de sumar (el valor inicial ya no sería 0).

RESPUESTA 16: b) 5 veces. El ciclo empieza con $i = 0$ y se repite mientras $i < 5$: los valores de i son 0, 1, 2, 3 y 4, es decir, cinco iteraciones. Cuando i llega a 5, la condición $5 < 5$ es falsa y el ciclo se detiene. Este es el error clásico de "uno de más o uno de menos": si cuentas los valores (0 a 4) en lugar de mirar solo el límite (5), no te equivocas.

Los errores de 'uno de más o uno de menos' (off-by-one) son legendarios en programación: hay un chiste que dice que los dos problemas más difíciles son nombrar variables, invalidar caché y los errores off-by-one. Para no caer en ellos, prueba tu ciclo con valores pequeños en papel: con ' $i < 5$ ' desde 0 obtienes 0,1,2,3,4 (cinco vueltas); con ' $i \leq 5$ ' obtendrías seis. Acostúmbrate a simular las primeras y últimas vueltas antes de dar el ciclo por bueno.

RESPUESTA 17: Falso. En un foreach, la variable de iteración es de solo lectura: el compilador no te permite asignarle un valor dentro del ciclo. Si intentas hacer algo como "numero = 0;" dentro de un foreach, obtendrás un error de compilación. Además, no puedes agregar ni quitar elementos de la colección mientras la recorres con foreach, porque eso lanza una excepción en tiempo de ejecución. Si necesitas modificar elementos, usa un for clásico con índices: "for (int i = 0; i < lista.Count; i++)".

Hay una segunda restricción del foreach igual de importante: no puedes agregar ni eliminar elementos de la colección mientras la recorres, porque obtendrás una excepción `InvalidOperationException` en tiempo de ejecución. Si necesitas filtrar o transformar, crea una lista nueva con los resultados (LINQ es perfecto para eso, como verás en la Unidad 7). Y si necesitas el índice además del valor, el foreach no te lo da: ahí el for clásico con 'lista[i]' es tu herramienta.

RESPUESTA 18:

```
static bool EsPar(int numero)
{
    return numero % 2 == 0;
}

Console.WriteLine(EsPar(8));
```

El método recibe un entero, usa el operador % para obtener el residuo de dividir entre 2 y devuelve true si el residuo es 0 (es par) o false si no lo es. El tipo de retorno es bool porque la pregunta que responde el método ("¿es par?") solo tiene dos respuestas posibles: sí o no. La última línea lo prueba con el 8 e imprime True en la consola (C# escribe los booleanos con mayúscula inicial: True y False).

Fíjate en el nombre del método: EsPar empieza con mayúscula y describe lo que pregunta, no lo que hace internamente. En C# los métodos se nombran con PascalCase (cada palabra con mayúscula inicial) y con verbos o preguntas claras: `CalcularTotal`, `GuardarArchivo`, `EsValido`. Un buen nombre te ahorra comentarios: cuando lees 'if (EsPar(numero))' entiendes la intención sin mirar el código del método. Nombrar bien es una habilidad profesional que se entrena desde el primer método que escribes.

RESPUESTA 19: c) 10. El método **Doble** multiplica por 2 lo que recibe: **Doble(4)** devuelve 8 y **Doble(1)** devuelve 2. La expresión "**Doble(4) + Doble(1)**" primero evalúa ambas llamadas (8 y 2) y luego suma los resultados: $8 + 2 = 10$. Fíjate en el orden: primero se ejecutan los métodos y se obtienen sus valores de retorno, y después se hace la suma. Los métodos se comportan como "fábricas" que entregan un valor listo para usarse en una expresión.

En los programas con top-level statements (como los de este libro), los métodos que escribes fuera de una clase deben llevar la palabra **static**, porque no pertenecen a ningún objeto: son funciones sueltas disponibles para todo el programa. Cuando trabajes con clases en la Unidad 5, verás métodos no estáticos que sí necesitan un objeto para ejecutarse ('miPerro.LadRAR()'). La diferencia entre **static** y no **static** es una de las ideas que más confunden al inicio, así que tenla presente.

RESPUESTA 20: Un parámetro es un DATO DE ENTRADA: es la información que el método necesita recibir para hacer su trabajo, y se declara entre los paréntesis en la definición del método. Un valor de retorno es el RESULTADO DE SALIDA: es lo que el método entrega de vuelta a quien lo llamó, y se declara con el tipo antes del nombre del método. Ejemplo:

```
static double CalcularArea(double radio)
{
    return 3.1416 * radio * radio;
}
```

Aquí "radio" es el parámetro (el dato que entra) y el área calculada es el valor de retorno (el dato que sale, de tipo **double**). La palabra **return** es la que envía el resultado de vuelta. Un método puede tener varios parámetros, uno solo o ninguno, y puede devolver un valor o nada (**void**).

No todos los métodos devuelven un valor: cuando un método solo **HACE** algo (imprimir, guardar, mostrar un menú) pero no entrega un resultado, su tipo de retorno es **void**. Ejemplo: 'static void MostrarSaludo(string nombre)' recibe el nombre como parámetro pero no devuelve nada. Y un método puede no tener parámetros pero sí devolver valor ('static int LeerOpcion()'), o no tener ni parámetros ni retorno. Las cuatro combinaciones existen y cada una tiene su uso: piénsalas como '¿necesita datos?' y '¿entrega un resultado?'.

RESPUESTA 21: b) int[] numeros = new int[5];. Los corchetes van después del tipo (**int[]**) para indicar "arreglo de enteros", y se usa **new int[5]** para crear el arreglo con espacio para 5 elementos. La opción a) está al revés: el tamaño no va en la declaración del tipo.

La opción c) usa una sintaxis que no existe en C# (parece de C++). La opción d) declara una simple variable int, no un arreglo. Recuerda: los 5 elementos se inicializan automáticamente en 0 y sus índices van del 0 al 4.

También puedes crear el arreglo con valores iniciales: `int[] numeros = { 1, 2, 3, 4, 5 };`. En ese caso no escribes el tamaño porque el compilador lo deduce de la cantidad de valores. Y recuerda que los elementos se inicializan en automático: un arreglo nuevo de int empieza lleno de ceros, uno de bool lleno de false y uno de string lleno de null. Nunca asumas que un arreglo 'vacío' tiene basura: en C# siempre empieza con los valores por defecto del tipo.

RESPUESTA 22: El programa imprime 7. Vamos por partes: `int[] n = { 2, 4, 6 };` crea un arreglo de 3 elementos donde `n[0]` es 2, `n[1]` es 4 y `n[2]` es 6. La expresión `n[1]` accede al segundo elemento (índice 1), que vale 4. La propiedad `n.Length` devuelve el tamaño del arreglo, que es 3. Entonces `n[1] + n.Length` es $4 + 3 = 7$. Esta pregunta combina dos ideas clave: los índices empiezan en 0 y `Length` te dice cuántos elementos hay.

No confundas `Length` con `Count`: `Length` es la propiedad de los arreglos (tamaño fijo) y `Count` es la de las listas y colecciones (tamaño variable). Si escribes `n.Count` sobre un arreglo, el compilador se quejará porque no existe. Esta diferencia te ayuda a recordar qué tipo de colección tienes entre manos. Además, `Length` en un arreglo multidimensional tiene sus trucos (devuelve el total de elementos), pero eso lo dominarás practicando en la Unidad 5.

RESPUESTA 23: Falso. En C# (como en la mayoría de los lenguajes modernos) los índices de un arreglo empiezan en 0, no en 1. El primer elemento se accede con `arreglo[0]`, el segundo con `arreglo[1]`, y el último con `arreglo[arreglo.Length - 1]`. Si intentas acceder a `arreglo[arreglo.Length]` obtendrás una excepción `IndexOutOfRangeException` porque ese índice no existe. La razón histórica es que el índice representa el desplazamiento desde el inicio del arreglo.

De este detalle sale una fórmula que usarás miles de veces: el último elemento de un arreglo siempre está en el índice `'arreglo.Length - 1'`. Si tu arreglo tiene 5 elementos (índices 0 a 4), el último es `arreglo[4]`, que es `arreglo[5 - 1]`. Cuando recorras un arreglo con `for`, la condición correcta es `'i < arreglo.Length'` (con `<` estricto, no `<=`): así `i` llega máximo a `Length - 1` y nunca te sales del arreglo. Memoriza este patrón y te ahorrarás la mayoría de los `IndexOutOfRangeException`.

RESPUESTA 24:

```

class Perro
{
    public string nombre;

    public Perro(string nombre)
    {
        this.nombre = nombre;
    }

    public void Ladrar()
    {
        Console.WriteLine("Guau, soy " + nombre);
    }
}

Perro miPerro = new Perro("Firulais");
miPerro.Ladrar();

```

La clase define el molde: un atributo nombre para guardar el dato, un constructor que se ejecuta al crear el objeto y recibe el nombre (la palabra `this` distingue el atributo del parámetro, que se llaman igual), y el método `Ladrar` que usa el dato guardado. Las últimas dos líneas crean el objeto con `new` (ahí se ejecuta el constructor) y llaman al método con la sintaxis `objeto.Metodo()`. Al correrlo verás "Guau, soy Firulais".

Lo poderoso de las clases es que cada objeto guarda sus PROPIOS datos: si creas `'Perro fido = new Perro("Fido")'` además de `Firulais`, cada uno tiene su nombre independiente y llamar a `fido.Ladrar()` imprime `'Guau, soy Fido'`. Los objetos son como formularios impresos del mismo molde pero llenados con datos distintos. Esta independencia es la que permite modelar el mundo real: una lista de cien perros, cada uno con su nombre, sin escribir cien variables.

RESPUESTA 25: b) Un constructor es un método especial que se ejecuta automáticamente al crear un objeto y sirve para inicializar sus datos. Se reconoce porque se llama igual que la clase y no declara tipo de retorno (ni siquiera `void`). La opción a) describe un destructor, que en C# casi nunca se escribe manualmente. Gracias al constructor, puedes obligar a que todo objeto nazca con datos válidos: en el ejemplo del `Perro`, es imposible crear un perro sin nombre porque el constructor lo exige.

Si no escribes ningún constructor, C# te regala uno invisible: el constructor por defecto, sin parámetros, que crea el objeto con los valores por defecto (0, null, false). Por eso `'new Perro()'` compila aunque no hayas escrito constructores. Pero en cuanto escribes UN constructor con parámetros, el compilador retira el regalo: `'new Perro()'` dejará de compilar a menos que

también escribas el constructor sin parámetros. Si necesitas ambas formas, escribe los dos constructores.

RESPUESTA 26: b) private. Un miembro marcado como **private** solo puede usarse dentro de la clase donde fue declarado: ni las clases hijas ni el resto del programa pueden verlo. **public** es lo contrario (visible desde todas partes), **protected** lo hace visible también en las clases que heredan, e **internal** lo hace visible dentro del mismo proyecto o ensamblado. Esta es la base del encapsulamiento: esconder los detalles internos y exponer solo lo necesario.

La convención profesional es: campos privados, acceso mediante propiedades públicas. Es decir, casi nunca verás un campo **public** en código bien escrito, porque exponer el campo directamente impide validar o cambiar la implementación después. La combinación '**private**' + propiedad '**public**' te da lo mejor de los dos mundos: control total por dentro y uso cómodo por fuera. Cuando veas código con campos públicos, considéralo una señal de código descuidado o de ejemplo simplificado.

RESPUESTA 27: Verdadero. Una propiedad es una forma elegante y segura de exponer un dato: por fuera se usa como si fuera una variable pública (**objeto.Edad = 20**), pero por dentro tiene código que se ejecuta al leer (**get**) y al escribir (**set**), donde puedes validar.

Ejemplo:

```
private int edad;
public int Edad
{
    get { return edad; }
    set
    {
        if (value >= 0)
            edad = value;
    }
}
```

Aquí el campo **edad** es privado (nadie lo toca directamente) y la propiedad **Edad** controla la escritura: si alguien intenta asignar una edad negativa, el **set** la ignora gracias a la validación. La palabra **value** representa el valor que se intenta asignar. Así logras el encapsulamiento sin renunciar a la comodidad.

Para propiedades simples sin validación existe la forma corta: las propiedades autoimplementadas. En lugar de escribir el campo privado más el **get** y el **set** a mano, escribes '**public string Nombre { get; set; }**' y el compilador crea el campo oculto por ti. Es la forma que

usarás el 90% de las veces. Y puedes controlar cada lado por separado: '{ get; private set; }' permite leer desde fuera pero solo modificar desde dentro de la clase, ideal para datos que se fijan al crear el objeto.

RESPUESTA 28:

```
class Rectangulo
{
    private double ancho;
    private double alto;

    public double Ancho
    {
        get { return ancho; }
        set { ancho = value < 0 ? 0 : value; }
    }

    public double Alto
    {
        get { return alto; }
        set { alto = value < 0 ? 0 : value; }
    }

    public double CalcularArea()
    {
        return ancho * alto;
    }
}
```

Los campos son privados para que nadie los modifique directamente, y cada propiedad valida en su set: si el valor entrante (value) es negativo, guarda 0; si no, guarda el valor tal cual. El método CalcularArea multiplica base por altura y devuelve el resultado como double. Si pruebas "r.Ancho = -5;" y luego lees "r.Ancho", obtendrás 0, lo que demuestra que la validación funciona y que el objeto nunca queda en un estado inválido.

Cuando varias propiedades comparten la misma regla de validación, evita repetir el código: extrae la validación a un método privado y llámalo desde cada set. Por ejemplo, un método 'private double ValidarPositivo(double valor)' que devuelve 0 si el valor es negativo. Así, si la regla cambia (digamos que ahora el mínimo es 1 en lugar de 0), la corriges en un solo lugar. Este principio se llama DRY (Don't Repeat Yourself: no te repitas) y es una de las marcas de un programador profesional.

RESPUESTA 29: b) Hola B. Aunque la variable `obj` está declarada de tipo A, el objeto real que contiene es de tipo B (`new B()`). Como el método `Saludar` está marcado como virtual en A y `override` en B, C# usa enlace dinámico: en tiempo de ejecución revisa el tipo REAL del objeto y llama a la versión de B. Esto es el polimorfismo en su forma más pura: la misma llamada "`obj.Saludar()`" se comporta distinto según el objeto que haya detrás. Si el método no fuera virtual, se imprimiría "Hola A" porque se usaría el tipo de la variable, no el del objeto.

Cuidado con una trampa parecida: si en la clase hija escribes un método con el MISMO nombre pero sin `override` (y sin virtual en la base), no hay polimorfismo sino 'ocultamiento': el compilador te avisa con una advertencia y el método que se ejecuta depende del tipo de la VARIABLE, no del objeto. La palabra `override` es la que activa el enlace dinámico. En el examen, si ves 'virtual' + 'override', piensa polimorfismo (manda el objeto); si no los ves, piensa en el tipo declarado de la variable.

RESPUESTA 30: El polimorfismo significa "muchas formas": es la capacidad de que un mismo método se comporte de manera distinta según la clase del objeto que lo ejecuta. Se logra con herencia más métodos virtuales y `override`. Ejemplo: una clase base `Animal` con un método virtual `HacerSonido`, y dos clases hijas que lo sobrescriben:

```

class Animal
{
    public virtual void HacerSonido()
    {
        Console.WriteLine("Sonido genérico");
    }
}

class Perro : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("Guau");
    }
}

class Gato : Animal
{
    public override void HacerSonido()
    {
        Console.WriteLine("Miau");
    }
}

```

Si creas una lista de Animal con un Perro y un Gato y llamas a HacerSonido en cada uno, el perro ladra y el gato maúlla, aunque la variable sea de tipo Animal. El beneficio es enorme: puedes escribir código que trabaje con la clase base sin saber ni importar qué clases hijas existen, y cada hija aporta su propio comportamiento.

El polimorfismo brilla cuando lo combinas con colecciones: una 'List<Animal>' puede guardar perros, gatos y pájaros mezclados, y un solo foreach llamando a HacerSonido() produce el sonido correcto de cada uno sin un solo if. Imagina lo horrible que sería el código alternativo: preguntar con if qué tipo es cada animal para decidir qué sonido hacer. El polimorfismo elimina esos condicionales y hace que agregar un animal nuevo (una Vaca) no requiera tocar el código existente: solo creas la clase.

RESPUESTA 31: b) Una interfaz es un contrato: define qué métodos y propiedades debe tener una clase, pero sin implementar ninguno. La clase que "firma" el contrato (con dos puntos, como en "class Pajaro : IVolador") está obligada a escribir el código de cada miembro. La gracia es que varias clases sin relación entre sí pueden implementar la misma interfaz: un Pájaro y un Avión no comparten herencia, pero ambos pueden implementar IVolar. A diferencia de la herencia, una clase puede implementar varias interfaces a la vez.

Las interfaces en C# se nombran por convención con I mayúscula al inicio: IVolador, IComparable, IDisposable. Esta convención es tan fuerte que con solo ver la I sabes que hablas de una interfaz. Además, una clase puede implementar VARIAS interfaces a la vez ('class Pato : Animal, IVolador, INadador'), algo imposible con la herencia de clases, que solo permite UN padre. Por eso se dice que las interfaces te dan 'herencia múltiple de comportamiento' sin sus problemas.

RESPUESTA 32: El programa imprime 4. Línea por línea: "List<int> numeros = new List<int> { 5, 10, 15 };" crea una lista genérica de enteros con 3 elementos iniciales (a diferencia de los arreglos, la lista puede crecer). "numeros.Add(20);" agrega un cuarto elemento al final. "numeros.Count" devuelve cuántos elementos hay en ese momento: 4. Fíjate en la diferencia con los arreglos: la lista no tiene un tamaño fijo, crece sola con Add, y por eso existe la propiedad Count en lugar de Length.

La lista genérica tiene más trucos en la manga: Remove(valor) quita un elemento, RemoveAt(indice) quita por posición, Contains(valor) pregunta si existe, Clear() la vacía e Insert(indice, valor) mete un elemento en medio. Todas trabajan sobre la marcha, sin que declares tamaños. Practica combinándolas: crea una lista, agrega cinco números, quita uno, pregunta si contiene otro e imprime el Count al final. Dominar List<T> es dominar la colección que más vas a usar en tu vida profesional.

RESPUESTA 33: Verdadero. LINQ (Language Integrated Query) te permite escribir consultas directamente en C# con métodos como Where, que recibe una condición (normalmente una expresión lambda) y devuelve solo los elementos que la cumplen. Por ejemplo, "numeros.Where(n => n > 10)" revisa cada elemento y te entrega los mayores que 10, sin que escribas ningún for ni foreach manualmente. El ciclo sigue existiendo, pero lo ejecuta LINQ por ti detrás de escena. Esto hace el código más corto, más legible y con menos lugares donde equivocarte.

Un detalle fino de LINQ: la ejecución es diferida (lazy). Cuando escribes 'numeros.Where(...)', todavía NO se recorre la lista: solo se construye la consulta. El recorrido real ocurre cuando materializas el resultado con ToList() o cuando lo recorres con foreach. Esto significa que si

modificas la lista entre la consulta y su ejecución, el resultado reflejará los datos nuevos. Es eficiente y elegante, pero puede sorprender: si necesitas una 'foto' inmediata de los datos, usa `ToList()` en ese momento.

RESPUESTA 34:

```
List<int> numeros = new List<int> { 4, 25, 8, 30, 12, 3 };  
List<int> resultado = numeros.Where(n => n > 10).OrderBy(n => n).ToList();
```

La consulta encadena tres operaciones: `Where(n => n > 10)` filtra y se queda solo con los mayores que 10 (25, 30, 12), `OrderBy(n => n)` los ordena de menor a mayor (12, 25, 30), y `ToList()` convierte el resultado en una lista concreta. La expresión "`n => n > 10`" es una lambda: se lee "dado n, devuelve si n es mayor que 10". LINQ se lee casi como español: "donde n sea mayor que 10, ordénalos y pásalos a lista".

LINQ tiene dos sintaxis equivalentes: la de métodos (la que usamos aquí, con `Where` y `OrderBy` encadenados) y la de consultas, que se parece a SQL: 'from n in numeros where n > 10 orderby n select n'. Ambas compilan a lo mismo; la de métodos es más popular en código profesional moderno. Mi consejo: domina primero la sintaxis de métodos porque es la que verás en documentación, tutoriales y entrevistas de trabajo, y reconoce la sintaxis de consultas cuando la encuentres.

RESPUESTA 35: b) Crea una nueva lista que contiene solo los números pares. El método `Where` recorre la lista original y selecciona los elementos que cumplen la condición "`n % 2 == 0`" (residuo cero al dividir entre 2): el 2, el 4 y el 6. `ToList()` materializa esa selección en una lista nueva e independiente. Importante: la lista original no se modifica (sigue teniendo los 6 números) y no se suma nada: `Where` filtra, no acumula. Si quisieras la suma usarías `Sum()`.

Fíjate en que 'var pares' usa `var`: el tipo real es `List<int>`, pero escribirlo sería redundante porque `ToList()` ya deja claro qué devuelve. Este es el caso de uso ideal de `var` que mencionamos en la Unidad 2. Otro punto: las lambdas como '`n => n % 2 == 0`' son la puerta de entrada a la programación funcional en C#. La flecha '`=>`' se lee como 'tal que' o 'va hacia': 'n tal que n módulo 2 sea cero'. Cuando las domines, `Where`, `Select` y `OrderBy` se volverán tu forma favorita de procesar datos.

RESPUESTA 36: c) finally. El bloque try contiene el código riesgoso, los bloques catch atrapan tipos específicos de excepciones, y finally se ejecuta siempre: si todo salió bien, si hubo excepción atrapada e incluso si hubo una excepción no atrapada. Por eso es el lugar ideal para liberar recursos, como cerrar un archivo o una conexión, porque garantiza que esa limpieza ocurra pase lo que pase. Un try puede llevar finally sin ningún catch, y eso es válido.

Un try puede llevar finally SIN ningún catch: 'try { ... } finally { ... }'. Esto es válido y útil cuando no quieres manejar el error aquí (dejas que suba a quien llamó), pero sí necesitas garantizar la limpieza de recursos. También puedes tener varios catch para distintos tipos de excepción, del más específico al más general: primero FileNotFoundException, luego IOException, luego Exception. El orden importa: un catch general antes que uno específico hace que el específico nunca se alcance.

RESPUESTA 37: El programa imprime "Error de formato". La línea "int.Parse("abc")" intenta convertir el texto "abc" en un número entero, lo cual es imposible, así que lanza una excepción de tipo FormatException. En ese instante se interrumpe el try y el programa salta directamente al bloque catch que atrapa FormatException, donde se imprime el mensaje. La línea "Conversión exitosa" nunca se ejecuta porque la excepción ocurrió antes. Este es el patrón básico del manejo de errores: intentas algo riesgoso y tienes un plan B si falla.

La forma profesional de evitar esta excepción es no usar Parse con datos del usuario, sino int.TryParse, que devuelve false en lugar de lanzar excepción cuando el texto no es número. Las excepciones son para situaciones EXCEPCIONALES (el archivo desapareció a mitad de la lectura), no para controlar el flujo normal del programa (el usuario escribió mal un número). Abusar del try-catch para validaciones comunes hace el código lento y difícil de leer: valida primero, atrapa después.

RESPUESTA 38: Verdadero. File.WriteAllText tiene dos comportamientos: si el archivo no existe, lo crea y escribe el contenido; si ya existe, borra todo su contenido anterior y escribe el nuevo texto encima. Por eso hay que usarlo con cuidado: es destructivo. Si lo que quieres es AGREGAR texto al final sin borrar lo que ya hay, el método correcto es File.AppendAllText. Ambos pertenecen a la clase estática File del espacio de nombres System.IO y escriben el archivo completo en una sola llamada.

Si necesitas escribir muchas líneas, no llames a WriteAllText cien veces: es ineficiente porque abre y cierra el archivo en cada llamada. Mejor acumula el texto (por ejemplo con StringBuilder) y escribe una sola vez, o usa un StreamWriter dentro de un bloque 'using', que mantiene el archivo abierto y lo cierra automáticamente al terminar. Y recuerda: estas

operaciones pueden lanzar excepciones (carpeta sin permisos, disco lleno), así que en programas reales van dentro de try-catch.

RESPUESTA 39:

```
try
{
    string contenido = File.ReadAllText("datos.txt");
    Console.WriteLine(contenido);
}
catch (FileNotFoundException)
{
    Console.WriteLine("El archivo no existe");
}
```

File.ReadAllText lee el archivo completo de una sola vez y lo devuelve como un string. Si el archivo no existe, este método lanza FileNotFoundException, que atrapamos en el catch para mostrar un mensaje amable en lugar de que el programa truene con un error técnico. Para que File funcione necesitas "using System.IO;" al inicio del archivo. Nota: también podrías atrapar IOException (más general), pero atrapar el tipo específico es mejor práctica porque tu plan B es preciso.

Nota sobre la ruta: 'datos.txt' es una ruta relativa, lo que significa que el programa busca el archivo en la carpeta actual de ejecución (normalmente la carpeta del proyecto al usar 'dotnet run'). Si el archivo está en otro lado, usa la ruta completa o muévelo junto al proyecto. Para saber dónde está buscando tu programa, puedes imprimir 'Directory.GetCurrentDirectory()'. Este detalle de las rutas relativas causa más dolores de cabeza a principiantes que cualquier excepción: cuando 'el archivo no existe' pero juras que sí, casi siempre es la carpeta.

RESPUESTA 40: b) "HOLA MUNDO". El método ToUpper() devuelve una copia del texto con todas las letras convertidas a mayúsculas. Importante: no modifica el texto original (los strings en C# son inmutables), sino que devuelve un texto nuevo; si no lo guardas en una variable o lo imprimes directamente, el resultado se pierde. Los métodos ToLower(), Trim() (quita espacios de los extremos) y Contains() (revisa si contiene un fragmento) siguen la misma idea y son tus herramientas básicas para trabajar con texto.

Que los strings sean inmutables tiene una consecuencia práctica: si concatenas texto dentro de un ciclo miles de veces con +, en cada vuelta se crea un string nuevo y el rendimiento se desploma. Para esos casos existe StringBuilder (en System.Text), que acumula el texto en un búfer mutable y solo al final produces el string con ToString(). Regla simple: unas pocas concatenaciones con + están bien; un ciclo que arma un texto grande pide StringBuilder. Es un detalle que distingue al programador que piensa en rendimiento.

RESPUESTA 41: b) Marcar un método con async le permite usar await en su interior para esperar operaciones lentas (descargas, lectura de archivos, acceso a datos) sin bloquear el hilo mientras espera. Un método async normalmente devuelve Task o Task<T> (la "promesa" de un resultado futuro) en lugar de devolver el valor directamente. Ojo con la opción c): async no hace nada más rápido por arte de magia; su ventaja es que el programa sigue respondiendo mientras espera, no la velocidad.

Cuidado con un error clásico: 'async void'. Un método async debería devolver Task o Task<T> casi siempre; usar 'async void' solo es aceptable en manejadores de eventos, porque las excepciones lanzadas en un 'async void' no se pueden atrapar con try-catch y pueden tumbar tu aplicación. Si tu método async no devuelve ningún valor útil, declara que devuelve 'Task' a secas: 'static async Task GuardarAsync()'. Así quien lo llame puede esperarlo con await y atrapar sus errores.

RESPUESTA 42: Falso. await no congela el programa: lo que hace es suspender ESE método y liberar el hilo para que pueda hacer otro trabajo (atender la interfaz, procesar otras tareas), y cuando la operación termina, el método continúa justo donde se quedó. Es una espera sin bloqueo, muy distinta a Thread.Sleep, que sí bloquea el hilo por completo. Gracias a esto, una aplicación puede descargar un archivo y seguir respondiendo a los clics del usuario al mismo tiempo.

El ejemplo clásico es una aplicación con interfaz gráfica: sin await, descargar un archivo congelaría la ventana y el usuario pensaría que el programa se trabó; con await, la ventana sigue respondiendo a clics mientras la descarga avanza en segundo plano. En aplicaciones de consola la diferencia se nota menos, pero el principio es el mismo y en ASP.NET (el framework web de .NET) es crítico: un servidor que bloquea hilos atendiendo descargas se queda sin hilos para otros usuarios. Por eso async/await es obligatorio en el desarrollo profesional moderno.

RESPUESTA 43:

```
static async Task<string> DescargarDatos()
{
    await Task.Delay(2000);
    return "descarga lista";
}

// Para usarlo:
string resultado = await DescargarDatos();
Console.WriteLine(resultado);
```

El método lleva `async`, devuelve `Task<string>` (la promesa de un texto futuro) y usa `await Task.Delay(2000)` para simular una espera de 2000 milisegundos sin bloquear el hilo. Después del `await`, devuelve el texto con `return` normal: el `await` se encarga de desenvolver la tarea y entregar el string a quien llamó al método. Para poder usar `await` al llamarlo, el código que llama también debe estar en un método `async` (en programas modernos de consola, los top-level statements ya permiten `await` directo).

Si necesitas lanzar varias operaciones a la vez y esperarlas todas, no las esperes una por una (eso las haría secuenciales): guárdalas en variables `Task` y usa '`await Task.WhenAll(tarea1, tarea2)`'. Así las dos corren en paralelo y el `await` termina cuando la última completa. Es el patrón estándar para descargar varios archivos o consultar varias fuentes de datos al mismo tiempo. Y si solo te interesa la primera que termine, existe `Task.WhenAny`. La asincronía bien usada convierte esperas muertas en trabajo paralelo.

RESPUESTA 44: a) Un delegado es un tipo que representa una referencia a un método: una variable de tipo delegado puede guardar un método y llamarlo después como si fuera el método mismo. Se declaran con la palabra `delegate` indicando la firma (parámetros y retorno) que deben tener los métodos compatibles. Su uso más famoso son los eventos: un botón no sabe qué código quieres ejecutar al hacer clic, así que expone un evento y tú le "conectas" tu método mediante un delegado. En la práctica moderna casi siempre se usan los delegados genéricos ya existentes: `Func` (devuelve valor) y `Action` (no devuelve nada).

En el C# moderno rara vez declaras tus propios delegados con la palabra '`delegate`', porque ya existen dos genéricos que cubren casi todo: '`Func<int, bool>`' representa un método que recibe un `int` y devuelve un `bool`, y '`Action<string>`' representa un método que recibe un `string` y no devuelve nada. LINQ está construido sobre ellos: el '`Where(n => n > 10)`' recibe justamente un '`Func<int, bool>`'. Entender delegados es entender el enchufe que conecta tus lambdas con métodos como `Where`, `Select` y `OrderBy`.

RESPUESTA 45: Una clase genérica es una clase que trabaja con un tipo de dato que se decide al momento de usarla, representado por un parámetro de tipo como `T`. `List<T>` es el ejemplo perfecto: la misma clase sirve como `List<int>` (lista de enteros), `List<string>` (lista de textos) o `List<Estudiante>` (lista de tus propios objetos), sin reescribir código.

Usar `List<int>` es mejor que una lista de object por dos razones principales. Primera, seguridad de tipos en compilación: en una `List<int>` el compilador te impide meter un texto por accidente, mientras que en una lista de object todo entra y el error aparece hasta que el programa corre. Segunda, rendimiento y comodidad: los tipos de valor como `int` sufren "boxing" (empaquetado) al guardarse como object y "unboxing" al leerse, lo que cuesta tiempo y memoria; además, al leer de una lista de object tienes que convertir (cast) manualmente cada elemento, mientras que `List<int>` te devuelve `int` directamente, listo para operar.

Los genéricos también aceptan restricciones con la palabra 'where': puedes exigir que `T` sea una clase, que tenga un constructor sin parámetros o que implemente cierta interfaz ('where `T : IComparable`'). Esto te permite escribir código genérico pero con garantías: por ejemplo, un método que ordena cualquier lista cuyos elementos sepan compararse. Sin restricciones, dentro de una clase genérica solo puedes hacer con `T` lo que se puede hacer con cualquier object. Las restricciones son el punto medio entre flexibilidad total y seguridad total.

RESPUESTA 46: b) Separar las responsabilidades en clases con nombres claros, cada una en su propio archivo. Esta es la base del código mantenible: si cada clase hace una sola cosa bien definida (una clase `Producto` para los datos, una clase `Inventario` para la lógica, la clase `Program` solo para el menú), encontrar y arreglar errores es mucho más fácil que cazarlos en un archivo gigante. Las otras opciones son malas prácticas clásicas: el archivo gigante se vuelve inmanejable, los nombres de una letra no dicen nada (usa "contador" en lugar de "c"), y evitar métodos solo duplica código.

En proyectos reales, cada archivo de clase suele empezar con la declaración de su 'namespace' (espacio de nombres), que organiza las clases en grupos lógicos y evita choques de nombres: 'namespace `MiTienda`;' agrupa todo lo de tu tienda. Los namespaces son como los apellidos de las clases: puede haber dos clases 'Producto' si viven en namespaces distintos. Mientras tus proyectos son pequeños no los extrañarás, pero en cuanto el proyecto crezca a decenas de archivos, serán tu sistema de organización.

RESPUESTA 47: Una estructura limpia para el inventario sería de tres clases. Primero, la clase `Producto` con los datos de cada artículo: `Nombre (string)`, `Precio (double)` y `Cantidad (int)`, más un constructor para crearlos con datos iniciales y quizás un método `MostrarDatos` que devuelva el producto como texto listo para guardar. Segundo, la clase `Inventario` que administra la colección: guarda una `List<Producto>`, y tiene los métodos

AgregarProducto, ListarProductos, BuscarPorNombre, más GuardarEnArchivo (escribe cada producto como una línea en "inventario.txt" con File.WriteAllText) y CargarDesdeArchivo (lee el archivo al iniciar y reconstruye la lista). Tercero, la clase Program que solo contiene el menú: muestra las opciones, lee la elección del usuario y llama a los métodos del Inventario. Así cada clase tiene una responsabilidad única: Producto guarda datos, Inventario maneja la lógica y el archivo, Program habla con el usuario. Si mañana quieres cambiar el archivo por una base de datos, solo tocas la clase Inventario.

Un buen hábito al diseñar estas clases es probar cada una por separado antes de conectarlas: crea unos productos de prueba, agrégalos al inventario en memoria, lista y busca, y solo cuando eso funcione conecta el guardado en archivo. Este enfoque incremental (probar por capas) hace que los errores sean fáciles de localizar: si algo falla, sabes en qué capa está. Los programadores profesionales trabajan así todos los días: nunca escriben mil líneas de un jalón esperando que todo funcione a la primera.

RESPUESTA 48: Verdadero. Nunca debes confiar en que el usuario escribirá lo que esperas: siempre validas antes de procesar. Ejemplo concreto: tu programa pide la edad con "int edad = Convert.ToInt32(Console.ReadLine());". Si el usuario escribe "veinte" en letras, la conversión lanza FormatException y el programa se rompe en su cara. La versión correcta valida primero, por ejemplo con int.TryParse, que intenta convertir y te dice si tuvo éxito sin lanzar excepción:

```
Console.Write("Escribe tu edad: ");
if (int.TryParse(Console.ReadLine(), out int edad))
{
    Console.WriteLine("Tienes " + edad + " años");
}
else
{
    Console.WriteLine("Eso no es un número válido");
}
```

Validar la entrada es la diferencia entre un programa de juguete y un programa profesional: el profesional anticipa al usuario distraído, no le echa la culpa.

La validación tiene dos niveles: el tipo (¿es un número?) y el rango (¿tiene sentido?). Un usuario podría escribir 250 como edad: es un número válido, pero una edad imposible. La validación completa revisa ambas cosas: 'if (edad >= 0 && edad <= 120)'. Este segundo nivel es lógica de tu negocio, no del lenguaje, y es lo que separa un programa que 'no se rompe' de uno que 'funciona bien'. Acostúmbrate a preguntar siempre: ¿qué valores tienen sentido aquí, además de ser del tipo correcto?

RESPUESTA 49: El programa imprime:

```
Ana aprobó
Luis reprobó
Marta aprobó
```

Vamos línea por línea: la clase Estudiante tiene dos propiedades automáticas (Nombre y Calificacion) y el método Aprobo, que devuelve true si la calificación es mayor o igual a 6. Se crea una lista con tres estudiantes: Ana (9), Luis (4) y Marta (7). El foreach recorre la lista: para Ana, Aprobo() devuelve true (9 >= 6) e imprime "Ana aprobó"; para Luis devuelve false (4 < 6) e imprime "Luis reprobó"; para Marta devuelve true (7 >= 6) e imprime "Marta aprobó". Este ejercicio integra cuatro temas del libro: clases con propiedades, métodos que devuelven bool, listas genéricas y foreach con condicionales.

Fíjate en la sintaxis 'new Estudiante { Nombre = "Ana", Calificacion = 9 }': se llama inicializador de objetos y te permite crear el objeto y asignar sus propiedades en una sola expresión, sin llamar a un constructor con parámetros. Es azúcar sintáctica muy usada en C# moderno, sobre todo para crear listas de objetos de prueba como esta. Por dentro equivale a crear el objeto vacío y asignar cada propiedad línea por línea, pero se lee mucho mejor.

RESPUESTA 50:

```
int opcion;

do
{
    Console.WriteLine("=== MENÚ ===");
    Console.WriteLine("1. Agregar producto");
    Console.WriteLine("2. Listar productos");
    Console.WriteLine("3. Buscar producto");
    Console.WriteLine("4. Salir");
    Console.Write("Elige una opción: ");
    int.TryParse(Console.ReadLine(), out opcion);

    switch (opcion)
    {
```

```

        case 1:
            AgregarProducto();
            break;
        case 2:
            ListarProductos();
            break;
        case 3:
            BuscarProducto();
            break;
        case 4:
            Console.WriteLine("Hasta luego");
            break;
        default:
            Console.WriteLine("Opción no válida");
            break;
    }
} while (opcion != 4);

static void AgregarProducto()
{
    Console.WriteLine("Agregando producto...");
}

static void ListarProductos()
{
    Console.WriteLine("Lista de productos...");
}

static void BuscarProducto()
{
    Console.WriteLine("Buscando producto...");
}

```

El do-while garantiza que el menú se muestre al menos una vez y se repita hasta que el usuario elija 4. El switch dirige cada opción a su método con su break obligatorio, y el case default atrapa opciones inválidas. Fíjate en que se usa `int.TryParse` en lugar de `Convert.ToInt32`: así, si el usuario escribe letras, `opcion` queda en 0, cae en default y el menú se repite sin romperse. Este es el esqueleto de casi cualquier programa de consola con menú: domínalo y podrás construir encima el inventario, el sistema de calificaciones o el juego de la Unidad 10.

Este menú es un patrón que vas a reutilizar en cada proyecto de consola: do-while + switch + métodos por opción. Para hacerlo aún más profesional, extrae la impresión del menú a su propio método 'MostrarMenu()' que devuelva la opción elegida, y deja el do-while solo como el

'motor' que repite. Así el código principal queda limpio y cada pieza tiene un nombre. Cuando domines este esqueleto, agregar la opción 5 o conectar el menú con archivos y clases será cuestión de minutos, no de horas.

Curso Completo de C#

Libro de programación de Lucas EGA Academy: explicaciones extensas unidad por unidad, práctica guiada y examen final para dominar el lenguaje desde cero.

Colibrí Editorial EGA

lucasegaacademy.com