Chapter 1. Introducing C# and .NET

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language is programmer productivity. To this end, C# balances simplicity, expressiveness, and performance. The chief architect of the language since its first version is Anders Hejlsberg (creator of Turbo Pascal and architect of Delphi). The C# language is platform neutral and works with a range of platform-specific runtimes.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation means creating a boundary around an *object* to separate its external (public) behavior from its internal (private) implementation details. Following are the distinctive features of C# from an object-oriented perspective:

Unified type system

> The fundamental building block in C# is an encapsulated unit of data and functions called a *type*. C# has a *unified type system* in which all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic functionality. For example, an instance of any type can be converted to a string by calling its `ToString` method.

Classes and interfaces

In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an *interface*. An interface is like a class that cannot hold data. This means that it can define only *behavior* (and not *state*), which allows for multiple inheritance as well as a separation between specification and implementation.

Properties, methods, and events

In the pure object-oriented paradigm, all functions are *methods*. In C#, methods are only one kind of *function member*, which also includes *properties* and *events* (there are others, too). Properties are function members that encapsulate a piece of an object's state such as a button's color or a label's text. Events are function members that simplify acting on object state changes.

Although C# is primarily an object-oriented language, it also borrows from the *functional programming* paradigm, specifically:

Functions can be treated as values

Using *delegates*, C# allows functions to be passed as values to and from other functions.

C# supports patterns for purity

Core to functional programming is avoiding the use of variables whose values change, in favor of declarative patterns. C# has key features to help with those patterns,

including the ability to write unnamed functions on the fly that "capture" variables (*lambda expressions*), and the ability to perform list or reactive programming via *query expressions*. C# also provides *records*, which make it easy to write *immutable* (read-only) types.

Type Safety

C# is primarily a *type-safe* language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents you from interacting with a *string* type as though it were an *integer* type.

More specifically, C# supports *static typing*, meaning that the language enforces type safety at *compile time*. This is in addition to type safety being enforced at *runtime*.

Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program because it knows for a given variable what type it is, and hence what methods you can call on that variable. Such tools can also identify everywhere in your program that a variable, type, or method is used, allowing for reliable refactoring.

> **NOTE**
>
> C# also allows parts of your code to be dynamically typed via the `dynamic` keyword. However, C# remains a predominantly statically typed language.

C# is also called a *strongly typed language* because its type rules are strictly enforced (whether statically or at runtime). For instance, you cannot call a function that's designed to accept an integer with a floating-point number, unless you first *explicitly* convert the floating-point number to an integer. This helps prevent mistakes.

Memory Management

C# relies on the runtime to perform automatic memory management. The Common Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++.

C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks. For performance-critical hotspots and interoperability, pointers and explicit memory allocation is permitted in blocks that are marked `unsafe`.

Platform Support

C# has runtimes that support the following platforms:

- Windows 7+ Desktop (for rich-client, web, server, and command-line applications)
- macOS (for web and command-line applications—and rich-client applications via Mac Catalyst)
- Linux (for web and command-line applications)
- Android and iOS (for mobile applications)
- Windows 10 devices (Xbox, Surface Hub, and HoloLens) via UWP

There is also a technology called *Blazor* that can compile C# to web assembly that runs in a browser.

CLRs, BCLs, and Runtimes

Runtime support for C# programs consists of a *Common Language Runtime* and a *Base Class Library*. A runtime can also include a higher-level *application layer* that contains libraries for developing rich-client, mobile, or web applications (see Figure 1-1). Different runtimes exist to allow for different kinds of applications, as well as different platforms.

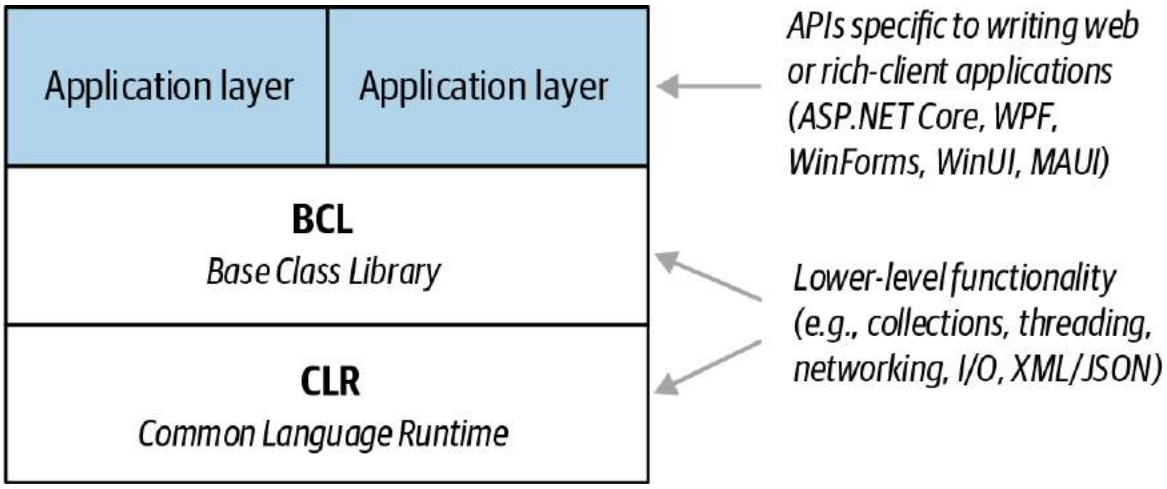

Figure 1-1. Runtime architecture

Common Language Runtime

A *Common Language Runtime* (CLR) provides essential runtime services such as automatic memory management and exception handling. (The word "common" refers to the fact that the same runtime can be shared by other *managed* programming languages, such as F#, Visual Basic, and Managed C++.)

C# is called a *managed language* because it compiles source code into managed code, which is represented in *Intermediate Language* (IL). The CLR converts the IL into the native code of the machine, such as X64 or X86, usually just prior to execution. This is referred to as Just-In-Time (JIT)

compilation. Ahead-of-time compilation is also available to improve startup time with large assemblies or resource-constrained devices (and to satisfy iOS app store rules when developing mobile apps).

The container for managed code is called an *assembly*. An assembly contains not only IL but also type information (*metadata*). The presence of metadata allows assemblies to reference types in other assemblies without needing additional files.

> **NOTE**
>
> You can examine and disassemble the contents of an assembly with Microsoft's *ildasm* tool. And with tools such as ILSpy or JetBrain's dotPeek, you can go further and decompile the IL to C#. Because IL is higher level than native machine code, the decompiler can do quite a good job of reconstructing the original C#.

A program can query its own metadata (*reflection*) and even generate new IL at runtime (*reflection.emit*).

Base Class Library

A CLR always ships with a set of assemblies called a *Base Class Library* (BCL). A BCL provides core functionality to programmers, such as collections, input/output, text processing, XML/JSON handling, networking, encryption, interop, concurrency, and parallel programming.

A BCL also implements types that the C# language itself requires (for features such as enumeration, querying, and asynchrony) and lets you explicitly access features of the CLR, such as Reflection and memory management.

Runtimes

A *runtime* (also called a *framework*) is a deployable unit that you download and install. A runtime consists of a CLR (with its BCL), plus an optional *application layer* specific to the kind of application that you're writing—

web, mobile, rich client, etc. (If you're writing a command-line console application or a non-UI library, you don't need an application layer.)

When writing an application, you *target* a particular runtime, which means that your application uses and depends on the functionality that the runtime provides. Your choice of runtime also determines which platforms your application will support.

The following table lists the major runtime options:

Application layer	CLR/BCL	Program type	Runs on...
ASP.NET	.NET 8	Web	Windows, Linux, macOS
Windows Desktop	.NET 8	Windows	Windows 10+
WinUI 3	.NET 8	Windows	Windows 10+
MAUI	.NET 8	Mobile, desktop	iOS, Android, macOS, Windows 10+
.NET Framework	.NET Framework	Web, Windows	Windows 7+

Figure 1-2 shows this information graphically and also serves as a guide to what's covered in the book.

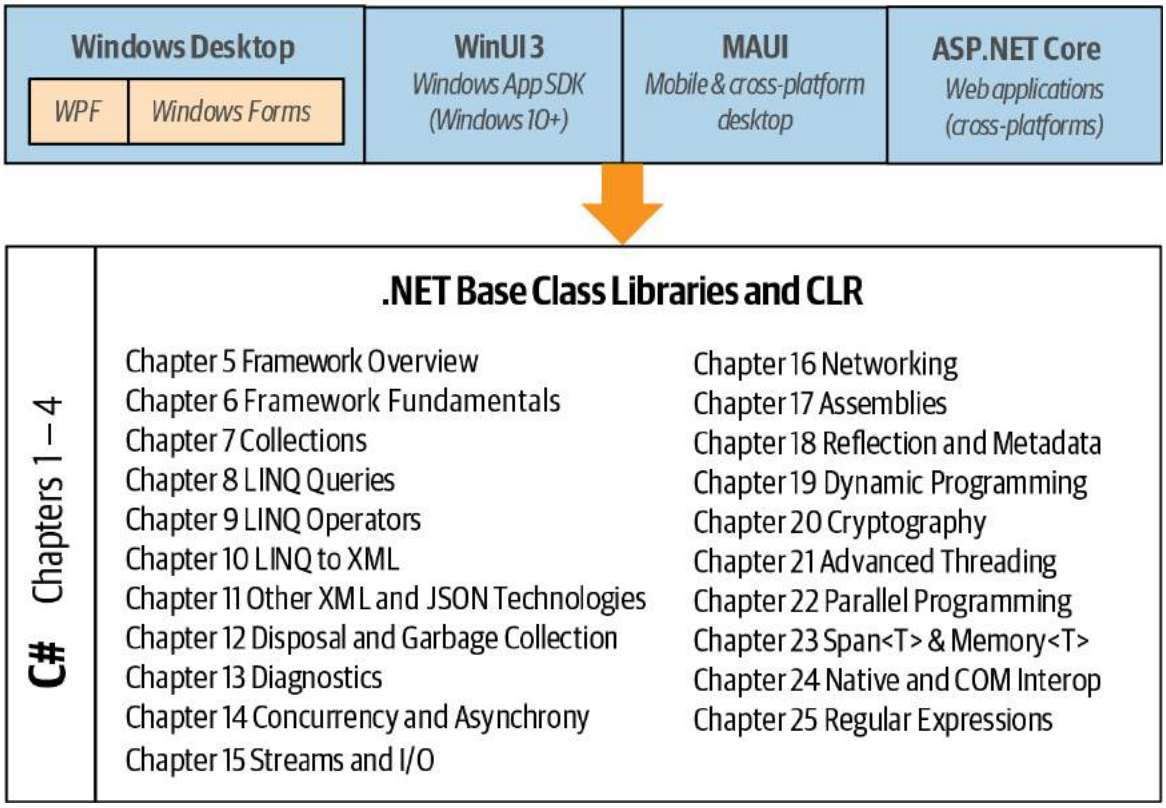

Figure 1-2. Runtimes for C#

.NET 8

.NET 8 is Microsoft's flagship open-source runtime. You can write web and console applications that run on Windows, Linux, and macOS; rich-client applications that run on Windows 10+ and macOS; and mobile apps that run on iOS and Android. This book focuses on the .NET 8 CLR and BCL.

Unlike .NET Framework, .NET 8 is not preinstalled on Windows machines. If you try to run a .NET 8 application without the correct runtime being present, a message will appear directing you to a web page where you can download the runtime. You can avoid this by creating a *self-contained* deployment, which includes the parts of the runtime required by the application.

NOTE

.NET's update history runs as follows: .NET Core 1.x → NET Core 2.x → .NET Core 3.x → .NET 5 → .NET 6 → .NET 7 → .NET 8. After .NET Core 3, Microsoft removed "Core" from the name and skipped version 4 to avoid confusion with *.NET Framework* 4.x, which precedes all of the preceding runtimes but is still supported and in popular use.

This means that assemblies compiled under .NET Core 1.x → .NET 7 will, in most cases, run without modification under .NET 8. In contrast, assemblies compiled under (any version of) .NET Framework are usually incompatible with .NET 8.

Windows Desktop and WinUI 3

For writing rich-client applications that run on Windows 10 and above, you can choose between the classic Windows Desktop APIs (Windows Forms and WPF) and WinUI 3. The Windows Desktop APIs are part of the .NET Desktop runtime, whereas WinUI 3 is part of the *Windows App SDK* (a separate download).

The classic Windows Desktop APIs have existed since 2006 and enjoy terrific third-party library support, as well as offering a wealth of answered questions on sites such as StackOverflow. *WinUI 3* was released in 2022 and is intended for writing modern immersive applications that feature the latest Windows 10+ controls. It is a successor to the *Universal Windows Platform* (UWP).

MAUI

MAUI (Multi-platform App UI) is designed primarily for creating mobile apps for iOS and Android, although it can also be used for desktop apps that run on macOS and Windows via Mac Catalyst and WinUI 3. MAUI is an evolution of Xamarin and allows a single project to target multiple platforms.

> **NOTE**
>
> For cross-platform desktop applications, a third-party library called Avalonia offers an alternative to MAUI. Avalonia also runs on Linux and is architecturally simpler than MAUI (as it operates without the Catalyst/WinUI indirection layer). Avalonia has an API similar to WPF, and it also offers a commercial add-on called XPF that provides almost complete WPF compatibility.

.NET Framework

.NET Framework is Microsoft's original Windows-only runtime for writing web and rich-client applications that run (only) on Windows desktop/server. No major new releases are planned, although Microsoft will continue to support and maintain the current 4.8 release due to the wealth of existing applications.

With the .NET Framework, the CLR/BCL is integrated with the application layer. Applications written in .NET Framework can be recompiled under .NET 8, although they usually require some modification. Some features of .NET Framework are not present in .NET 8 (and vice versa).

.NET Framework is preinstalled with Windows and is automatically patched via Windows Update. When you target .NET Framework 4.8, you can use the features of C# 7.3 and earlier. (You can override this by specifying a newer language version in the project file—this unlocks all of the latest language features except for those that require support from a newer runtime.)

> **NOTE**
>
> The word ".NET" has long been used as an umbrella term for any technology that includes the word ".NET" (.NET Framework, .NET Core, .NET Standard, and so on).
>
> This means that Microsoft's renaming of .NET Core to .NET has created an unfortunate ambiguity. In this book, we'll refer to the new .NET as *.NET 5+* when an ambiguity arises. And to refer to .NET Core and its successors, we'll use the phrase ".NET Core and .NET 5+."
>
> To add to the confusion, .NET (5+) is a framework, yet it's very different from the *.NET Framework*. Hence, we'll use the term *runtime* in preference to *framework*, where possible.

Niche Runtimes

There are also the following niche runtimes:

- Unity is a game development platform that allows game logic to be scripted with C#.
- *Universal Windows Platform* (UWP) was designed for writing touch-first applications that run on Windows 10+ desktop and devices, including Xbox, Surface Hub, and HoloLens. UWP apps are sandboxed and ship via the Windows Store. UWP uses a version of the .NET Core 2.2 CLR/BCL, and it's unlikely that this dependency will be updated; instead, Microsoft has recommended that users switch to its modern replacement, WinUI 3. But because WinUI 3 only supports Windows desktop, UWP still has a niche application for targeting Xbox, Surface Hub, and HoloLens.
- The .NET Micro Framework is for running .NET code on highly resource-constrained embedded devices (under one megabyte).

It's also possible to run managed code within SQL Server. With SQL Server CLR integration, you can write custom functions, stored procedures, and aggregations in C# and then call them from SQL. This works in conjunction with .NET Framework and a special "hosted" CLR that enforces a sandbox to protect the integrity of the SQL Server process.

A Brief History of C#

The following is a reverse chronology of the new features in each C# version, for the benefit of readers who are already familiar with an older version of the language.

What's New in C# 12

C# 12 ships with Visual Studio 2022, and is used when you target .NET 8.

Collection expressions

Rather than initializing an array as follows:

```
char[] vowels = {'a','e','i','o','u'};
```

you can now use square brackets (a *collection expression*):

```
char[] vowels = ['a','e','i','o','u'];
```

Collection expressions have two major advantages. First, the same syntax also works with other collection types, such as lists and sets (and even the low-level span types):

```
List<char> list         = ['a','e','i','o','u'];
HashSet<char> set       = ['a','e','i','o','u'];
ReadOnlySpan<char> span = ['a','e','i','o','u'];
```

Second, they are *target-typed*, which means that you can omit the type in other scenarios where the compiler can infer it, such as when calling methods:

```
Foo (['a','e','i','o','u']);

void Foo (char[] letters) { ... }
```

See "Collection Initializers and Collection Expressions" for more details.

Primary constructors in classes and structs

From C# 12, you can include a parameter list directly after a class (or struct) declaration:

```
class Person (string firstName, string lastName)
{
  public void Print() => Console.WriteLine (firstName + " " + lastName);
}
```

This instructs the compiler to automatically build a *primary constructor*, allowing the following:

```
Person p = new Person ("Alice", "Jones");
p.Print();    // Alice Jones
```

This feature has existed since C# 9 with records—where they behave slightly differently. With records, the compiler generates (by default) a public init-only property for each primary constructor parameter. This is not the case with classes and structs; to achieve the same result, you must define those properties explicitly:

```
class Person (string firstName, string lastName)
{
  public string FirstName { get; set; } = firstName;
  public string LastName { get; set; } = lastName;
}
```

Primary constructors work well in simple scenarios. We describe their nuances and limitations in "Primary Constructors (C# 12)".

Default lambda parameters

Just as ordinary methods can define parameters with default values:

```
void Print (string message = "") => Console.WriteLine (message);
```

so, too, can lambda expressions:

```
var print = (string message = "") => Console.WriteLine (message);

print ("Hello");
print ();
```

This feature is useful with libraries such as ASP.NET Minimal API.

Alias any type

C# has always allowed you to alias a simple or generic type via the `using` directive:

```
using ListOfInt = System.Collections.Generic.List<int>;
```

```
var list = new ListOfInt();
```

From C# 12, this approach works with other kinds of types, too, such as arrays and tuples:

```
using NumberList = double[];
using Point = (int X, int Y);

NumberList numbers = { 2.5, 3.5 };
Point p = (3, 4);
```

Other new features

C# 12 also supports *inline arrays*, via the `[System.Runtime.CompilerServices.InlineArray]` attribute. This allows for the creation of fixed-size arrays in a struct without requiring an unsafe context, and is intended for use primarily within the runtime APIs.

What's New in C# 11

C# 11 shipped with Visual Studio 2022, and is used by default when you target .NET 7.

Raw string literals

Wrapping a string in three or more quote characters creates a *raw string literal*, which can contain almost any character sequence without escaping or doubling up. This makes it easy to represent JSON, XML, and HTML literals, as well as regular expressions and source code:

```
string raw = """<file path="c:\temp\test.txt"></file>""";
```

Raw string literals can be multiline and permit interpolation via the `$` prefix:

```
string multiLineRaw = $"""
  Line 1
  Line 2
```

```
    The date and time is {DateTime.Now}
    """;
```

Using two (or more) `$` characters in a raw string literal prefix changes the interpolation sequence from one brace to two (or more) braces, allowing you to include braces in the string itself:

```
Console.WriteLine ($$"""{ "TimeStamp": "{{DateTime.Now}}" }""");
// Output: { "TimeStamp": "01/01/2024 12:13:25 PM" }
```

We cover the nuances of this feature in "Raw string literals (C# 11)" and "String interpolation".

UTF-8 strings

With the `u8` suffix, you create string literals encoded in UTF-8 rather than UTF-16. This feature is intended for advanced scenarios such as the low-level handling of JSON text in performance hotspots:

```
ReadOnlySpan<byte> utf8 = "ab→cd"u8;  // Arrow symbol consumes 3 bytes
Console.WriteLine (utf8.Length);      // 7
```

The underlying type is `ReadOnlySpan<byte>` (Chapter 23), which you can convert to a byte array by calling its `ToArray()` method.

List patterns

List patterns match a series of elements in square brackets, and work with any collection type that is countable (with a `Count` or `Length` property) and indexable (with an indexer of type `int` or `System.Index`):

```
int[] numbers = { 0, 1, 2, 3, 4 };
Console.WriteLine (numbers is [0, 1, 2, 3, 4]);   // True
```

An underscore matches a single element of any value, and two dots match zero or more elements (a *slice*):

```
Console.WriteLine (numbers is [_, 1, .., 4]);     // True
```

A slice can be followed by the `var` pattern—see "List Patterns" for details.

Required members

Applying the `required` modifier to a field or property forces consumers of that class or struct to populate that member via an object initializer when constructing it:

```
Asset a1 = new Asset { Name = "House" };  // OK
Asset a2 = new Asset();                   // Error: will not compile!

class Asset { public required string Name; }
```

With this feature, you can avoid writing constructors with long parameter lists, which can simplify subclassing. Should you also wish to write a constructor, you can apply the `[SetsRequiredMembers]` attribute to bypass the required member restriction for that constructor—see "Required members (C# 11)" for details.

Static virtual/abstract interface members

From C# 11, interfaces can declare members as `static virtual` or `static abstract`:

```
public interface IParsable<TSelf>
{
   static abstract TSelf Parse (string s);
}
```

These members are implemented as static functions in classes or structs, and can be called polymorphically via a constrained type parameter:

```
T ParseAny<T> (string s) where T : IParsable<T> => T.Parse (s);
```

Operator functions can also be declared as `static virtual` or `static abstract`.

For details, see "Static virtual/abstract interface members" and "Static Polymorphism". We also describe how to call static abstract members via reflection in "Calling Static Virtual/Abstract Interface Members".

Generic math

The `System.Numerics.INumber<TSelf>` interface (new to .NET 7) unifies arithmetic operations across all numeric types, allowing generic methods such as the following to be written:

```
T Sum<T> (T[] numbers) where T : INumber<T>
{
  T total = T.Zero;
  foreach (T n in numbers)
    total += n;      // Invokes addition operator for any numeric type
  return total;
}

int intSum = Sum (3, 5, 7);
double doubleSum = Sum (3.2, 5.3, 7.1);
decimal decimalSum = Sum (3.2m, 5.3m, 7.1m);
```

`INumber<TSelf>` is implemented by all real and integral numeric types in .NET (as well as `char`), and comprises several interfaces that include static abstract operator definitions such as the following:

```
static abstract TResult operator + (TSelf left, TOther right);
```

We cover this in "Polymorphic Operators" and "Generic Math".

Other new features

A type with the `file` accessibility modifier can be accessed only from within the same file, and is intended for use within source generators:

```
file class Foo { ... }
```

C# 11 also introduced checked operators (see "Checked operators"), for defining operator functions to be called inside `checked` blocks (this was

required for a full implementation of generic math). C# 11 also relaxed the requirement to populate every field in a struct's constructor (see "Struct Construction Semantics").

Finally, the `nint` and `nuint` native-sized integers types that were introduced in C# 9 to match the address space of the process at runtime (32 or 64 bits) were enhanced in C# 11 when targeting .NET 7 or later. Specifically, the compile-time distinction between these types and their underlying runtime types (`IntPtr` and `UIntPtr`) has melted away when targeting .NET 7+. See "Native-Sized Integers" for a full discussion.

What's New in C# 10

C# 10 shipped with Visual Studio 2022, and is used when you target .NET 6.

File-scoped namespaces

In the common case that all types in a file are defined in a single namespace, a *file-scoped namespace* declaration in C# 10 reduces clutter and eliminates an unnecessary level of indentation:

```
namespace MyNamespace;  // Applies to everything that follows in the file.

class Class1 {}         // inside MyNamespace
class Class2 {}         // inside MyNamespace
```

The global using directive

When you prefix a `using` directive with the `global` keyword, it applies the directive to all files in the project:

```
global using System;
global using System.Collection.Generic;
```

This lets you avoid repeating the same directives in every file. `global using` directives work with `using static`.

Additionally, .NET 6 projects now support *implicit global using directives*: if the `ImplicitUsings` element is set to true in the project file, the most commonly used namespaces are automatically imported (based on the SDK project type). See "The global using Directive" for more detail.

Nondestructive mutation for anonymous types

C# 9 introduced the `with` keyword, to perform nondestructive mutation on records. In C# 10, the `with` keyword also works with anonymous types:

```
var a1 = new { A = 1, B = 2, C = 3, D = 4, E = 5 };
var a2 = a1 with { E = 10 };
Console.WriteLine (a2);      // { A = 1, B = 2, C = 3, D = 4, E = 10 }
```

New deconstruction syntax

C# 7 introduced the deconstruction syntax for tuples (or any type with a `Deconstruct` method). C# 10 takes this syntax further, letting you mix assignment and declaration in the same deconstruction:

```
var point = (3, 4);
double x = 0;
(x, double y) = point;
```

Field initializers and parameterless constructors in structs

From C# 10, you can include field initializers and parameterless constructors in structs (see "Structs"). These execute only when the constructor is called explicitly, and so can easily be bypassed—for instance, via the `default` keyword. This feature was introduced primarily for the benefit of struct records.

Record structs

Records were first introduced in C# 9, where they acted as a compiled-enhanced class. In C# 10, records can also be structs:

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

The rules are otherwise similar: *record structs* have much the same features as *class structs* (see "Records"). An exception is that the compiler-generated properties on record structs are writable, unless you prefix the record declaration with the `readonly` keyword.

Lambda expression enhancements

The syntax around lambda expressions has been enhanced in a number of ways. First, implicit typing (`var`) is permitted:

```
var greeter = () => "Hello, world";
```

The implicit type for a lambda expression is an `Action` or `Func` delegate, so `greeter`, in this case, is of type `Func<string>`. You must explicitly state any parameter types:

```
var square = (int x) => x * x;
```

Second, a lambda expression can specify a return type:

```
var sqr = int (int x) => x;
```

This is primarily to improve compiler performance with complex nested lambdas.

Third, you can pass a lambda expression into a method parameter of type `object`, `Delegate`, or `Expression`:

```
M1 (() => "test");   // Implicitly typed to Func<string>
M2 (() => "test");   // Implicitly typed to Func<string>
M3 (() => "test");   // Implicitly typed to Expression<Func<string>>

void M1 (object x) {}
void M2 (Delegate x) {}
void M3 (Expression x) {}
```

Finally, you can apply attributes to a lambda expression's compile-generated target method (as well as its parameters and return value):

```
Action a = [Description("test")] () => { };
```

See "Applying Attributes to Lambda Expressions" for more detail.

Nested property patterns

The following simplified syntax is legal in C# 10 for nested property pattern matching (see "Property Patterns"):

```
var obj = new Uri ("https://www.linqpad.net");
if (obj is Uri { Scheme.Length: 5 }) ...
```

This is equivalent to:

```
if (obj is Uri { Scheme: { Length: 5 }}) ...
```

CallerArgumentExpression

A method parameter to which you apply the `[CallerArgumentExpression]` attribute captures an argument expression from the call site:

```
Print (Math.PI * 2);

void Print (double number,
            [CallerArgumentExpression("number")] string expr = null)
  => Console.WriteLine (expr);

// Output: Math.PI * 2
```

This feature is intended primarily for validation and assertion libraries (see "CallerArgumentExpression").

Other new features

The `#line` directive has been enhanced in C# 10 to allow a column and range to be specified.

Interpolated strings in C# 10 can be constants, as long as the interpolated values are constants.

Records can seal the `ToString()` method in C# 10.

C#'s definite assignment analysis has been improved so that expressions such as the following work:

```
if (foo?.TryParse ("123", out var number) ?? false)
  Console.WriteLine (number);
```

(Prior to C# 10, the compiler would generate an error: "Use of unassigned local variable 'number'.")

What's New in C# 9.0

C# 9.0 shipped with *Visual Studio 2019*, and is used when you target .NET 5.

Top-level statements

With *top-level statements* (see "Top-Level Statements"), you can write a program without the baggage of a `Main` method and `Program` class:

```
using System;
Console.WriteLine ("Hello, world");
```

Top-level statements can include methods (which act as local methods). You can also access command-line arguments via the "magic" `args` variable, and return a value to the caller. Top-level statements can be followed by type and namespace declarations.

Init-only setters

An *init-only setter* (see "Init-only setters") in a property declaration uses the `init` keyword instead of the `set` keyword:

```
class Foo { public int ID { get; init; } }
```

This behaves like a read-only property, except that it can also be set via an object initializer:

```
var foo = new Foo { ID = 123 };
```

This makes it possible to create immutable (read-only) types that can be populated via an object initializer instead of a constructor, and helps to avoid the antipattern of constructors that accept a large number of optional parameters. Init-only setters also allow for *nondestructive mutation* when used in *records*.

Records

A *record* (see "Records") is a special kind of class that's designed to work well with immutable data. Its most special feature is that it supports *nondestructive mutation* via a new keyword (`with`):

```
Point p1 = new Point (2, 3);
Point p2 = p1 with { Y = 4 };   // p2 is a copy of p1, but with Y set to 4
Console.WriteLine (p2);         // Point { X = 2, Y = 4 }

record Point
{
  public Point (double x, double y) => (X, Y) = (x, y);

  public double X { get; init; }
  public double Y { get; init; }
}
```

In simple cases, a record can also eliminate the boilerplate code of defining properties and writing a constructor and deconstructor. We can replace our `Point` record definition with the following, without loss of functionality:

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

Like tuples, records exhibit structural equality by default. Records can subclass other records, and can include the same constructs that classes can include. The compiler implements records as classes at runtime.

Pattern-matching improvements

The *relational pattern* (see "Patterns") allows the <, >, <=, and >= operators to appear in patterns:

```
string GetWeightCategory (decimal bmi) => bmi switch {
  < 18.5m => "underweight",
  < 25m => "normal",
  < 30m => "overweight",
  _ => "obese" };
```

With *pattern combinators*, you can combine patterns via three new keywords (`and`, `or`, and `not`):

```
bool IsVowel (char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';

bool IsLetter (char c) => c is >= 'a' and <= 'z'
                           or >= 'A' and <= 'Z';
```

As with the `&&` and `||` operators, `and` has higher precedence than `or`. You can override this with parentheses.

The `not` combinator can be used with the *type pattern* to test whether an object is (not) a type:

```
if (obj is not string) ...
```

Target-typed new expressions

When constructing an object, C# 9 lets you omit the type name when the compiler can infer it unambiguously:

```
System.Text.StringBuilder sb1 = new();
System.Text.StringBuilder sb2 = new ("Test");
```

This is particularly useful when the variable declaration and initialization are in different parts of your code:

```
class Foo
{
  System.Text.StringBuilder sb;
```

```
  public Foo (string initialValue) => sb = new (initialValue);
}
```

And in the following scenario:

```
MyMethod (new ("test"));
void MyMethod (System.Text.StringBuilder sb) { ... }
```

See "Target-Typed new Expressions" for more information.

Interop improvements

C# 9 introduces *function pointers* (see "Function Pointers" and "Callbacks with Function Pointers"). Their main purpose is to allow unmanaged code to call static methods in C# without the overhead of a delegate instance, with the ability to bypass the P/Invoke layer when the arguments and return types are *blittable* (represented identically on each side).

C# 9 also introduces the `nint` and `nuint` native-sized integer types (see "Native-Sized Integers"), which map at runtime to `System.IntPtr` and `System.UIntPtr`. At compile time, they behave like numeric types with support for arithmetic operations.

Other new features

Additionally, C# 9 now lets you:

- Override a method or read-only property such that it returns a more derived type (see "Covariant return types").
- Apply attributes to local functions (see "Attributes").
- Apply the `static` keyword to lambda expressions or local functions to ensure that you don't accidentally capture local or instance variables (see "Static lambdas").
- Make any type work with the `foreach` statement, by writing a `GetEnumerator` extension method.

- Define a *module initializer* method that executes once when an assembly is first loaded, by applying the `[ModuleInitializer]` attribute to a (static void parameterless) method.
- Use a "discard" (underscore symbol) as a lambda expression argument.
- Write *extended partial methods* that are mandatory to implement—enabling scenarios such as Roslyn's new *source generators* (see "Extended partial methods").
- Apply an attribute to methods, types, or modules to prevent local variables from being initialized by the runtime (see "[SkipLocalsInit]").

What's New in C# 8.0

C# 8.0 first shipped with *Visual Studio 2019*, and is still used today when you target .NET Core 3 or .NET Standard 2.1.

Indices and ranges

Indices and ranges simplify working with elements or portions of an array (or the low-level types `Span<T>` and `ReadOnlySpan<T>`).

Indices let you refer to elements relative to the *end* of an array by using the `^` operator. `^1` refers to the last element, `^2` refers to the second-to-last element, and so on:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement  = vowels [^1];   // 'u'
char secondToLast = vowels [^2];   // 'o'
```

Ranges let you "slice" an array by using the `..` operator:

```
char[] firstTwo =  vowels [..2];    // 'a', 'e'
char[] lastThree = vowels [2..];    // 'i', 'o', 'u'
char[] middleOne = vowels [2..3]    // 'i'
char[] lastTwo =   vowels [^2..];   // 'o', 'u'
```

C# implements indexes and ranges with the help of the `Index` and `Range` types:

```
Index last = ^1;
Range firstTwoRange = 0..2;
char[] firstTwo = vowels [firstTwoRange];   // 'a', 'e'
```

You can support indices and ranges in your own classes by defining an indexer with a parameter type of `Index` or `Range`:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this   [Index index] => words [index];
  public string[] this [Range range] => words [range];
}
```

For more information, see "Indices and Ranges".

Null-coalescing assignment

The `??=` operator assigns a variable only if it's null. Instead of

```
if (s == null) s = "Hello, world";
```

you can now write this:

```
s ??= "Hello, world";
```

Using declarations

If you omit the brackets and statement block following a `using` statement, it becomes a *using declaration*. The resource is then disposed when execution falls outside the *enclosing* statement block:

```
if (File.Exists ("file.txt"))
{
  using var reader = File.OpenText ("file.txt");
  Console.WriteLine (reader.ReadLine());
```

```
  ...
}
```

In this case, `reader` will be disposed when execution falls outside the `if` statement block.

Read-only members

C# 8 lets you apply the `readonly` modifier to a struct's *functions*, ensuring that if the function attempts to modify any field, a compile-time error is generated:

```
struct Point
{
  public int X, Y;
  public readonly void ResetX() => X = 0;  // Error!
}
```

If a `readonly` function calls a non-`readonly` function, the compiler generates a warning (and defensively copies the struct to avoid the possibility of a mutation).

Static local methods

Adding the `static` modifier to a local method prevents it from seeing the local variables and parameters of the enclosing method. This helps to reduce coupling and enables the local method to declare variables as it pleases, without risk of colliding with those in the containing method.

Default interface members

C# 8 lets you add a default implementation to an interface member, making it optional to implement:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (text);
}
```

This means that you can add a member to an interface without breaking implementations. Default implementations must be called explicitly through the interface:

```
((ILogger)new Logger()).Log ("message");
```

Interfaces can also define static members (including fields), which can be accessed from code inside default implementations:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (Prefix + text);
  static string Prefix = "";
}
```

Or from outside the interface unless restricted via an accessibility modifier on the static interface member (such as `private`, `protected`, or `internal`):

```
ILogger.Prefix = "File log: ";
```

Instance fields are prohibited. For more details, see "Default Interface Members".

Switch expressions

From C# 8, you can use `switch` in the context of an *expression*:

```
string cardName = cardNumber switch    // assuming cardNumber is an int
{
  13 => "King",
  12 => "Queen",
  11 => "Jack",
  _ => "Pip card"   // equivalent to 'default'
};
```

For more examples, see "Switch expressions".

Tuple, positional, and property patterns

C# 8 supports three new patterns, mostly for the benefit of switch statements/expressions (see "Patterns"). *Tuple patterns* let you switch on multiple values:

```
int cardNumber = 12; string suite = "spades";
string cardName = (cardNumber, suite) switch
{
  (13, "spades") => "King of spades",
  (13, "clubs") => "King of clubs",
  ...
};
```

Positional patterns allow a similar syntax for objects that expose a deconstructor, and *property patterns* let you match on an object's properties. You can use all of the patterns both in switches and with the `is` operator. The following example uses a *property pattern* to test whether `obj` is a string with a length of 4:

```
if (obj is string { Length:4 }) ...
```

Nullable reference types

Whereas *nullable value types* bring nullability to value types, *nullable reference types* do the opposite and bring (a degree of) *non-nullability* to reference types, with the purpose of helping to avoid `NullReferenceExceptions`. Nullable reference types introduce a level of safety that's enforced purely by the compiler in the form of warnings or errors when it detects code that's at risk of generating a `NullReferenceException`.

Nullable reference types can be enabled either at the project level (via the `Nullable` element in the *.csproj* project file) or in code (via the `#nullable` directive). After it's enabled, the compiler makes non-nullability the default: if you want a reference type to accept nulls, you must apply the `?` suffix to indicate a *nullable reference type*:

```
#nullable enable    // Enable nullable reference types from this point on
```

```
string s1 = null;   // Generates a compiler warning! (s1 is non-nullable)
string? s2 = null;  // OK: s2 is nullable reference type
```

Uninitialized fields also generate a warning (if the type is not marked as nullable), as does dereferencing a nullable reference type, if the compiler thinks a `NullReferenceException` might occur:

```
void Foo (string? s) => Console.Write (s.Length);  // Warning (.Length)
```

To remove the warning, you can use the *null-forgiving operator* (`!`):

```
void Foo (string? s) => Console.Write (s!.Length);
```

For a full discussion, see "Nullable Reference Types".

Asynchronous streams

Prior to C# 8, you could use `yield return` to write an *iterator*, or `await` to write an *asynchronous function*. But you couldn't do both and write an iterator that awaits, yielding elements asynchronously. C# 8 fixes this through the introduction of *asynchronous streams*:

```
async IAsyncEnumerable<int> RangeAsync (
  int start, int count, int delay)
{
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    yield return i;
  }
}
```

The `await foreach` statement consumes an asynchronous stream:

```
await foreach (var number in RangeAsync (0, 10, 100))
  Console.WriteLine (number);
```

For more information, see "Asynchronous Streams".

What's New in C# 7.x

C# 7.x was first shipped with Visual Studio 2017. C# 7.3 is still used today by Visual Studio 2019 when you target .NET Core 2, .NET Framework 4.6 to 4.8, or .NET Standard 2.0.

C# 7.3

C# 7.3 made minor improvements to existing features, such as enabling the use of the equality operators with tuples, improved overload resolution, and the ability to apply attributes to the backing fields of automatic properties:

```
[field:NonSerialized]
public int MyProperty { get; set; }
```

C# 7.3 also built on C# 7.2's advanced low-allocation programming features, with the ability to reassign *ref locals*, no requirement to pin when indexing `fixed` fields, and field initializer support with `stackalloc`:

```
int* pointer  = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc []    {1, 2, 3};
```

Notice that stack-allocated memory can be assigned directly to a `Span<T>`. We describe spans—and why you would use them—in Chapter 23.

C# 7.2

C# 7.2 added a new `private protected` modifier (the *intersection* of `internal` and `protected`), the ability to follow named arguments with positional ones when calling methods, and `readonly` structs. A `readonly` struct enforces that all fields are `readonly`, to aid in declaring intent and to allow the compiler more optimization freedom:

```
readonly struct Point
{
  public readonly int X, Y;   // X and Y must be readonly
}
```

C# 7.2 also added specialized features to help with micro-optimization and low-allocation programming: see "The in modifier", "Ref Locals", "Ref Returns", and "Ref Structs".

C# 7.1

From C# 7.1, you can omit the type when using the `default` keyword, if the type can be inferred:

```
decimal number = default;   // number is decimal
```

C# 7.1 also relaxed the rules for switch statements (so that you can pattern-match on generic type parameters), allowed a program's `Main` method to be asynchronous, and allowed tuple element names to be inferred:

```
var now = DateTime.Now;
var tuple = (now.Hour, now.Minute, now.Second);
```

Numeric literal improvements

Numeric literals in C# 7 can include underscores to improve readability. These are called *digit separators* and are ignored by the compiler:

```
int million = 1_000_000;
```

Binary literals can be specified with the `0b` prefix:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

Out variables and discards

C# 7 makes it easier to call methods that contain `out` parameters. First, you can now declare *out variables* on the fly (see "Out variables and discards"):

```
bool successful = int.TryParse ("123", out int result);
Console.WriteLine (result);
```

And when calling a method with multiple `out` parameters, you can *discard* ones you're uninterested in with the underscore character:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
Console.WriteLine (x);
```

Type patterns and pattern variables

You can also introduce variables on the fly with the `is` operator. These are called *pattern variables* (see "Introducing a pattern variable"):

```
void Foo (object x)
{
  if (x is string s)
    Console.WriteLine (s.Length);
}
```

The `switch` statement also supports type patterns, so you can switch on *type* as well as constants (see "Switching on types"). You can specify conditions with a `when` clause and also switch on the `null` value:

```
switch (x)
{
  case int i:
    Console.WriteLine ("It's an int!");
    break;
  case string s:
    Console.WriteLine (s.Length);    // We can use the s variable
    break;
  case bool b when b == true:        // Matches only when b is true
    Console.WriteLine ("True");
    break;
  case null:
    Console.WriteLine ("Nothing");
    break;
}
```

Local methods

A *local method* is a method declared within another function (see "Local methods"):

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));

  int Cube (int value) => value * value * value;
}
```

Local methods are visible only to the containing function and can capture local variables in the same way that lambda expressions do.

More expression-bodied members

C# 6 introduced the expression-bodied "fat-arrow" syntax for methods, read-only properties, operators, and indexers. C# 7 extends this to constructors, read/write properties, and finalizers:

```
public class Person
{
  string name;

  public Person (string name) => Name = name;

  public string Name
  {
    get => name;
    set => name = value ?? "";
  }

  ~Person () => Console.WriteLine ("finalize");
}
```

Deconstructors

C# 7 introduces the *deconstructor* pattern (see "Deconstructors"). Whereas a constructor typically takes a set of values (as parameters) and assigns them to fields, a *deconstructor* does the reverse and assigns fields back to a set of variables. We could write a deconstructor for the `Person` class in the preceding example as follows (exception handling aside):

```
public void Deconstruct (out string firstName, out string lastName)
{
  int spacePos = name.IndexOf (' ');
  firstName = name.Substring (0, spacePos);
  lastName = name.Substring (spacePos + 1);
}
```

Deconstructors are called with the following special syntax:

```
var joe = new Person ("Joe Bloggs");
var (first, last) = joe;          // Deconstruction
Console.WriteLine (first);        // Joe
Console.WriteLine (last);         // Bloggs
```

Tuples

Perhaps the most notable improvement to C# 7 is explicit *tuple* support (see "Tuples"). Tuples provide a simple way to store a set of related values:

```
var bob = ("Bob", 23);
Console.WriteLine (bob.Item1);   // Bob
Console.WriteLine (bob.Item2);   // 23
```

C#'s new tuples are syntactic sugar for using the `System.ValueTuple<…>` generic structs. But thanks to compiler magic, tuple elements can be named:

```
var tuple = (name:"Bob", age:23);
Console.WriteLine (tuple.name);     // Bob
Console.WriteLine (tuple.age);      // 23
```

With tuples, functions can return multiple values without resorting to `out` parameters or extra type baggage:

```
static (int row, int column) GetFilePosition() => (3, 10);

static void Main()
{
  var pos = GetFilePosition();
  Console.WriteLine (pos.row);      // 3
  Console.WriteLine (pos.column);   // 10
}
```

Tuples implicitly support the deconstruction pattern, so you can easily *deconstruct* them into individual variables:

```
static void Main()
{
  (int row, int column) = GetFilePosition();   // Creates 2 local variables
  Console.WriteLine (row);      // 3
  Console.WriteLine (column);   // 10
}
```

throw expressions

Prior to C# 7, `throw` was always a statement. Now it can also appear as an expression in expression-bodied functions:

```
public string Foo() => throw new NotImplementedException();
```

A `throw` expression can also appear in a ternary conditional expression:

```
string Capitalize (string value) =>
  value == null ? throw new ArgumentException ("value") :
  value == "" ? "" :
  char.ToUpper (value[0]) + value.Substring (1);
```

What's New in C# 6.0

C# 6.0, which shipped with *Visual Studio 2015*, features a new-generation compiler, completely written in C#. Known as project "Roslyn," the new compiler exposes the entire compilation pipeline via libraries, allowing you to perform code analysis on arbitrary source code. The compiler itself is open source, and the source code is available at *https://github.com/dotnet/roslyn*.

In addition, C# 6.0 features several minor but significant enhancements, aimed primarily at reducing code clutter.

The *null-conditional* ("Elvis") operator (see "Null Operators") avoids having to explicitly check for null before calling a method or accessing a

type member. In the following example, `result` evaluates to null instead of throwing a `NullReferenceException`:

```
System.Text.StringBuilder sb = null;
string result = sb?.ToString();      // result is null
```

Expression-bodied functions (see "Methods") allow methods, properties, operators, and indexers that comprise a single expression to be written more tersely, in the style of a lambda expression:

```
public int TimesTwo (int x) => x * 2;
public string SomeProperty => "Property value";
```

Property initializers (Chapter 3) let you assign an initial value to an automatic property:

```
public DateTime TimeCreated { get; set; } = DateTime.Now;
```

Initialized properties can also be read-only:

```
public DateTime TimeCreated { get; } = DateTime.Now;
```

Read-only properties can also be set in the constructor, making it easier to create immutable (read-only) types.

Index initializers (Chapter 4) allow single-step initialization of any type that exposes an indexer:

```
var dict = new Dictionary<int,string>()
{
  [3] = "three",
  [10] = "ten"
};
```

String interpolation (see "String Type") offers a succinct alternative to `string.Format`:

```
string s = $"It is {DateTime.Now.DayOfWeek} today";
```

Exception filters (see "try Statements and Exceptions") let you apply a condition to a catch block:

```
string html;
try
{
  html = await new HttpClient().GetStringAsync ("http://asef");
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
{
  ...
}
```

The `using static` (see "Namespaces") directive lets you import all the static members of a type so that you can use those members unqualified:

```
using static System.Console;
...
WriteLine ("Hello, world");  // WriteLine instead of Console.WriteLine
```

The `nameof` (Chapter 3) operator returns the name of a variable, type, or other symbol as a string. This avoids breaking code when you rename a symbol in Visual Studio:

```
int capacity = 123;
string x = nameof (capacity);   // x is "capacity"
string y = nameof (Uri.Host);   // y is "Host"
```

And finally, you're now allowed to `await` inside `catch` and `finally` blocks.

What's New in C# 5.0

C# 5.0's big new feature was support for *asynchronous functions* via two new keywords, `async` and `await`. Asynchronous functions enable *asynchronous continuations*, which make it easier to write responsive and thread-safe rich-client applications. They also make it easy to write highly concurrent and efficient I/O-bound applications that don't tie up a thread

resource per operation. We cover asynchronous functions in detail in Chapter 14.

What's New in C# 4.0

C# 4.0 introduced four major enhancements:

Dynamic binding (Chapters 4 and 19) defers *binding*—the process of resolving types and members—from compile time to runtime and is useful in scenarios that would otherwise require complicated reflection code. Dynamic binding is also useful when interoperating with dynamic languages and COM components.

Optional parameters (Chapter 2) allow functions to specify default parameter values so that callers can omit arguments, and *named arguments* allow a function caller to identify an argument by name rather than position.

Type variance rules were relaxed in C# 4.0 (Chapters 3 and 4), such that type parameters in generic interfaces and generic delegates can be marked as *covariant* or *contravariant*, allowing more natural type conversions.

COM interoperability (Chapter 24) was enhanced in C# 4.0 in three ways. First, arguments can be passed by reference without the `ref` keyword (particularly useful in conjunction with optional parameters). Second, assemblies that contain COM interop types can be *linked* rather than *referenced*. Linked interop types support type equivalence, avoiding the need for *Primary Interop Assemblies* and putting an end to versioning and deployment headaches. Third, functions that return COM Variant types from linked interop types are mapped to `dynamic` rather than `object`, eliminating the need for casting.

What's New in C# 3.0

The features added to C# 3.0 were mostly centered on *Language-Integrated Query* (LINQ) capabilities. LINQ enables queries to be written directly within a C# program and checked *statically* for correctness, and query both local collections (such as lists or XML documents) or remote data sources

(such as a database). The C# 3.0 features added to support LINQ comprised implicitly typed local variables, anonymous types, object initializers, lambda expressions, extension methods, query expressions, and expression trees.

Implicitly typed local variables (`var` keyword, Chapter 2) let you omit the variable type in a declaration statement, allowing the compiler to infer it. This reduces clutter as well as allowing *anonymous types* (Chapter 4), which are simple classes created on the fly that are commonly used in the final output of LINQ queries. You can also implicitly type arrays (Chapter 2).

Object initializers (Chapter 3) simplify object construction by allowing you to set properties inline after the constructor call. Object initializers work with both named and anonymous types.

Lambda expressions (Chapter 4) are miniature functions created by the compiler on the fly; they are particularly useful in "fluent" LINQ queries (Chapter 8).

Extension methods (Chapter 4) extend an existing type with new methods (without altering the type's definition), making static methods feel like instance methods. LINQ's query operators are implemented as extension methods.

Query expressions (Chapter 8) provide a higher-level syntax for writing LINQ queries that can be substantially simpler when working with multiple sequences or range variables.

Expression trees (Chapter 8) are miniature code Document Object Models (DOMs) that describe lambda expressions assigned to the special type `Expression<TDelegate>`. Expression trees make it possible for LINQ queries to execute remotely (e.g., on a database server) because they can be introspected and translated at runtime (e.g., into an SQL statement).

C# 3.0 also added automatic properties and partial methods.

Automatic properties (Chapter 3) cut the work in writing properties that simply `get`/`set` a private backing field by having the compiler do that work

automatically. *Partial methods* (Chapter 3) let an autogenerated partial class provide customizable hooks for manual authoring that "melt away" if unused.

What's New in C# 2.0

The big new features in C# 2 were generics (Chapter 3), nullable value types (Chapter 4), iterators (Chapter 4), and anonymous methods (the predecessor to lambda expressions). These features paved the way for the introduction of LINQ in C# 3.

C# 2 also added support for partial classes, static classes, and a host of minor and miscellaneous features such as the namespace alias qualifier, friend assemblies, and fixed-size buffers.

The introduction of generics required a new CLR (CLR 2.0), because generics maintain full type fidelity at runtime.