- There are two new read-only collection types, `FrozenDictionary<K,V>` and `FrozenSet<T>`. These are like the existing `ImmutableDictionary<K,V>` and `ImmutableHashSet<T>` types but are optimized purely for reading, without methods for nondestructive mutation (see "Frozen Collections").
- `RegEx` now supports `RegexOptions.NonBacktracking` to help avoid denial-of-service attacks with user-supplied expressions (see "RegexOptions"). The regular expressions engine is also now faster.
- Types for SHA-3 hashing are now available, subject to operating system support (see "Hash Algorithms in .NET").

The JSON serialization engine has also been improved, with new features and better performance.

Runtime Targets and TFMs

Within a project file, the `<TargetFramework>` element determines which runtime the project is built against (its *framework target* or *runtime target*) and is denoted by a *Target Framework Moniker* (TFM). Valid values include `net8.0`, `net7.0`, `net6.0`, `net5.0` (for .NET versions 8, 7, 6, and 5), `netcoreapp3.1` (for .NET Core 3.1), `net48` (for .NET Framework 4.8), and `netstandard2.0` (which we cover in the following section). For example, this is how you target .NET 8:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
<PropertyGroup>
```

You can target multiple runtimes by instead specifying a `<TargetFrameworks>` element (plural). Each TFM is separated by a semicolon:

```
<TargetFrameworks>net8.0;net48</TargetFrameworks>
```

When you multitarget, the compiler generates a separate output assembly for each target.

The runtime target is encoded into the output assembly via the `TargetFramework` attribute. An assembly can run on a newer (but not older) runtime than its target.

.NET Standard

The wealth of public libraries that are available on NuGet wouldn't be as valuable if they supported only .NET 8. When writing a library, you'll often want to support a variety of platforms and runtime versions. To achieve that goal without creating a separate build for each runtime (multitargeting), you must target the lowest common denominator. This is relatively easy if you wish to support only .NET 8's direct predecessors: for example, if your project targets .NET 6 (`net6.0`), your library will run on .NET 6, .NET 7, and .NET 8.

The situation becomes messier if you also want to support .NET Framework (or legacy runtimes such as Xamarin). The reason is that each of these runtimes has a CLR and BCL with overlapping features—no one runtime is a pure subset of the others.

.NET Standard solves this problem by defining artificial subsets that work across an entire range of runtimes. By targeting .NET Standard, you can easily write libraries with extensive reach.

> **NOTE**
>
> .NET Standard is not a runtime; it's merely a specification describing a minimum baseline of functionality (types and members) that guarantees compatibility with a certain set of runtimes. The concept is similar to C# interfaces: .NET Standard is like an interface that concrete types (runtimes) can implement.

.NET Standard 2.0

The most useful version is *.NET Standard 2.0*. A library that targets .NET Standard 2.0 instead of a specific runtime will run without modification on both modern .NET (.NET 8/7/6/5, down to .NET Core 2) and .NET Framework (4.6.1+). It also supports the legacy UWP (from 10.0.16299+) and Mono 5.4+ (the CLR/BCL used by older versions of Xamarin).

To target .NET Standard 2.0, add the following to your *.csproj* file:

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
<PropertyGroup>
```

Most of the APIs described in this book are supported by .NET Standard 2.0 (and of those that are not, most are available as NuGet packages).

Other .NET Standards

NET Standard 2.1 is a superset of .NET Standard 2.0 that supports (only) the following platforms:

- .NET Core 3+
- Mono 6.4+

.NET Standard 2.1 is not supported by any version of .NET Framework, making it much less useful than .NET Standard 2.0.

There are also older .NET Standards such as 1.1, 1.2, 1.3, and 1.6, whose compatibility extends to archaic runtimes such as .NET Core 1.0 or .NET Framework 4.5. The 1.x standards lack thousands of APIs that are present in 2.0 (including much of what we describe in this book) and are effectively defunct.

.NET Framework and .NET 8 Compatibility

Because .NET Framework has existed for so long, it's not uncommon to encounter libraries that are available *only* for .NET Framework (with no

.NET Standard, .NET Core, or .NET 8 equivalent). To help mitigate this situation, .NET 5+ and .NET Core projects are permitted to reference .NET Framework assemblies, with the following provisos:

- An exception is thrown should the .NET Framework assembly call an API that's unsupported.
- Nontrivial dependencies might (and often do) fail to resolve.

In practice, it's most likely to work in simple cases, such as an assembly that wraps an unmanaged DLL.

Reference Assemblies

When you target .NET Standard, your project implicitly references an assembly called *netstandard.dll*, which contains all of the allowable types and members for your chosen version of .NET Standard. This is called a *reference assembly* because it exists only for the benefit of the compiler and contains no compiled code. At runtime, the "real" assemblies are identified through assembly redirection attributes (the choice of assemblies will depend on which runtime and platform the assembly eventually runs on).

Interestingly, a similar thing happens when you target .NET 8. Your project implicitly references a set of reference assemblies whose types mirror what's in the runtime assemblies for the chosen .NET version. This helps with versioning and cross-platform compatibility, and also allows you to target a different .NET version than what is installed on your machine.

Runtime and C# Language Versions

By default, your project's runtime target determines which C# language version is used:

Runtime target	C# version
.NET 8	C# 12
.NET 7	C# 11
.NET 6	C# 10
.NET 5	C# 9
.NET Core 3.x & 2.x	C# 8
.NET Framework	C# 7.3
.NET Standard 2.0	C# 7.3

This is because later versions of C# include features that rely on types that were introduced in later runtimes.

You can override the language version in your project file with the `<LangVersion>` element. Using an older runtime (such as .NET Framework) with a later language version (such as C# 12) means that the language features that rely on newer .NET types will not work (although in some cases, you can define those types yourself, or import them from a NuGet package).

The CLR and BCL

Note to the indexer: Please skip everything in this section (apart from the Level 1 heading). Everything in this section is covered in more detail later in the book.

System Types

The most fundamental types live directly in the `System` namespace. These include C#'s built-in types; the `Exception` base class; the `Enum`, `Array`, and `Delegate` base classes; and `Nullable`, `Type`, `DateTime`, `TimeSpan`, and `Guid`. The `System` namespace also includes types for performing mathematical functions (`Math`), generating random numbers (`Random`), and converting between various types (`Convert` and `BitConverter`).

Chapter 6 describes these types as well as the interfaces that define standard protocols used across .NET for such tasks as formatting (`IFormattable`) and order comparison (`IComparable`).

The `System` namespace also defines the `IDisposable` interface and the `GC` class for interacting with the garbage collector, which we cover in Chapter 12.

Text Processing

The `System.Text` namespace contains the `StringBuilder` class (the editable or *mutable* cousin of `string`) and the types for working with text encodings, such as UTF-8 (`Encoding` and its subtypes). We cover this in Chapter 6.

The `System.Text.RegularExpressions` namespace contains types that perform advanced pattern-based search-and-replace operations; we describe these in Chapter 25.

Collections

.NET offers a variety of classes for managing collections of items. These include both list- and dictionary-based structures; they work in conjunction with a set of standard interfaces that unify their common characteristics. All collection types are defined in the following namespaces, covered in Chapter 7:

```
System.Collections              // Nongeneric collections
System.Collections.Generic      // Generic collections
```

```
System.Collections.Frozen       // High-performance read-only collections
System.Collections.Immutable    // General-purpose read-only collections
System.Collections.Specialized  // Strongly typed collections
System.Collections.ObjectModel  // Bases for your own collections
System.Collections.Concurrent   // Thread-safe collection (Chapter 22)
```

Querying

Language-Integrated Query (LINQ) allows you to perform type-safe queries over local and remote collections (e.g., SQL Server tables) and is described in Chapters 8, 9, and 10. A big advantage of LINQ is that it presents a consistent querying API across a variety of domains. The essential types reside in the following namespaces:

```
System.Linq                 // LINQ to Objects and PLINQ
System.Linq.Expressions     // For building expressions manually
System.Xml.Linq             // LINQ to XML
```

XML and JSON

XML and JSON are widely supported in .NET. Chapter 10 focuses entirely on LINQ to XML—a lightweight XML Document Object Model (DOM) that can be constructed and queried through LINQ. Chapter 11 covers the performant low-level XML reader/writer classes, XML schemas and stylesheets, and types for working with JSON:

```
System.Xml                // XmlReader, XmlWriter
System.Xml.Linq           // The LINQ to XML DOM
System.Xml.Schema         // Support for XSD
System.Xml.Serialization  // Declarative XML serialization for .NET types
System.Xml.XPath          // XPath query language
System.Xml.Xsl            // Stylesheet support

System.Text.Json          // JSON reader/writer and DOM
System.Text.Json.Nodes    // JsonNode API (DOM)
```

In the online supplement at *http://www.albahari.com/nutshell*, we cover the JSON serializer.

Diagnostics

In Chapter 13, we cover logging and assertion and describe how to interact with other processes, write to the Windows event log, and handle performance monitoring. The types for this are defined in and under `System.Diagnostics`.

Concurrency and Asynchrony

Many modern applications need to deal with more than one thing happening at a time. Since C# 5.0, this has become easier through asynchronous functions and high-level constructs such as tasks and task combinators. Chapter 14 explains all of this in detail, after starting with the basics of multithreading. Types for working with threads and asynchronous operations are in the `System.Threading` and `System.Threading.Tasks` namespaces.

Streams and Input/Output

.NET provides a stream-based model for low-level input/output (I/O). Streams are typically used to read and write directly to files and network connections, and can be chained or wrapped in decorator streams to add compression or encryption functionality. Chapter 15 describes the stream architecture as well as the specific support for working with files and directories, compression, pipes, and memory-mapped files. The `Stream` and I/O types are defined in and under the `System.IO` namespace.

Networking

You can directly access most standard network protocols such as HTTP, TCP/IP, and SMTP via the types in `System.Net`. In Chapter 16, we demonstrate how to communicate using each of these protocols, starting with simple tasks such as downloading from a web page and finishing with using TCP/IP directly to retrieve POP3 email. Here are the namespaces we cover:

```
System.Net
System.Net.Http          // HttpClient
System.Net.Mail          // For sending mail via SMTP
System.Net.Sockets       // TCP, UDP, and IP
```

Assemblies, Reflection, and Attributes

The assemblies into which C# programs compile comprise executable instructions (stored as IL) and metadata, which describes the program's types, members, and attributes. Through reflection, you can inspect this metadata at runtime and do such things as dynamically invoke methods. With `Reflection.Emit`, you can construct new code on the fly.

In Chapter 17, we describe the makeup of assemblies and how to dynamically load and isolate them. In Chapter 18, we cover reflection and attributes—describing how to inspect metadata, dynamically invoke functions, write custom attributes, emit new types, and parse raw IL. The types for using reflection and working with assemblies reside in the following namespaces:

```
System
System.Reflection
System.Reflection.Emit
```

Dynamic Programming

In Chapter 19, we look at some of the patterns for dynamic programming and utilizing the Dynamic Language Runtime (DLR). We describe how to implement the *Visitor* pattern, write custom dynamic objects, and interoperate with IronPython. The types for dynamic programming are in `System.Dynamic`.

Cryptography

.NET provides extensive support for popular hashing and encryption protocols. In Chapter 20, we cover hashing, symmetric and public-key

encryption, and the Windows Data Protection API. The types for this are defined in:

```
System.Security
System.Security.Cryptography
```

Advanced Threading

C#'s asynchronous functions make concurrent programming significantly easier because they lessen the need for lower-level techniques. However, there are still times when you need signaling constructs, thread-local storage, reader/writer locks, and so on. Chapter 21 explains this in depth. Threading types are in the `System.Threading` namespace.

Parallel Programming

In Chapter 22, we cover in detail the libraries and types for leveraging multicore processors, including APIs for task parallelism, imperative data parallelism, and functional parallelism (PLINQ).

Span<T> and Memory<T>

To help with micro-optimizing performance hotspots, the CLR provides a number of types to help you program in such a way as to reduce the load on the memory manager. Two of the key types are `Span<T>` and `Memory<T>`, which we describe in Chapter 23.

Native and COM Interoperability

You can interoperate with both native and Component Object Model (COM) code. Native interoperability allows you to call functions in unmanaged DLLs, register callbacks, map data structures, and interoperate with native data types. COM interoperability allows you to call COM types (on Windows machines) and expose .NET types to COM. The types that support these functions are in `System.Runtime.InteropServices`, and we cover them in Chapter 24.

Regular Expressions

In Chapter 25, we cover how you can use regular expressions to match character patterns in strings.

Serialization

.NET providers several systems for saving and restoring objects to a binary or text representation. Such systems can be used for communication as well as saving and restoring objects to a file. In the online supplement at *http://www.albahari.com/nutshell*, we cover all four serialization engines: the binary serializer, the (newly updated) JSON serializer, the XML serializer, and the data contract serializer.

The Roslyn Compiler

The C# compiler itself is written in C#—the project is called "Roslyn," and the libraries are available as NuGet packages. With these libraries, you can utilize the compiler's functionality in many ways besides compiling source code to an assembly, such as writing code analysis and refactoring tools. We cover Roslyn in the online supplement, at *http://www.albahari.com/nutshell*.

Application Layers

User interface (UI)–based applications can be divided into two categories: *thin client*, which amounts to a website, and *rich client*, which is a program the end user must download and install on a computer or mobile device.

For writing thin-client applications in C#, there's ASP.NET Core, which runs on Windows, Linux, and macOS. ASP.NET Core is also designed for writing web APIs.

For rich-client applications, there is a choice of APIs:

- The Windows Desktop layer includes the popular WPF and Windows Forms APIs, and runs on Windows 7/8/10/11 desktop.
- WinUI 3 (Windows App SDK) is a successor to UWP that runs (only) on Windows 10+ desktop.
- UWP lets you write Windows Store apps that run on Windows 10+ desktop and devices such as Xbox or HoloLens.
- MAUI (formerly Xamarin) runs on iOS and Android mobile devices. MAUI also allows for cross-platform desktop applications that target macOS (via Catalyst) and Windows (via Windows App SDK).

There are also third-party cross-platform UI libraries such as Avalonia. Unlike MAUI, Avalonia also runs on Linux and does not rely on a Catalyst/WinUI indirection layer for desktop platforms, simplifying development and debugging.

ASP.NET Core

ASP.NET Core is a lightweight modular successor to ASP.NET and is suitable for creating web sites, REST-based web APIs, and microservices. It can also run in conjunction with two popular single-page-application frameworks: React and Angular.

ASP.NET supports the popular *Model-View-Controller* (MVC) pattern, as well as a newer technology called Blazor, where client-side code is written in C# instead of JavaScript.

ASP.NET Core runs on Windows, Linux, and macOS and can self-host in a custom process. Unlike its .NET Framework predecessor (ASP.NET), ASP.NET Core is not dependent on `System.Web` and the historical baggage of web forms.

As with any thin-client architecture, ASP.NET Core offers the following general advantages over rich clients:

- There is zero deployment at the client end.

- The client can run on any platform that supports a web browser.
- Updates are easily deployed.

Windows Desktop

The Windows Desktop application layer offers a choice of two UI APIs for writing rich-client applications: WPF and Windows Forms. Both APIs run on Windows Desktop/Server 7 through 11.

WPF

WPF was introduced in 2006, and has been enhanced ever since. Unlike its predecessor, Windows Forms, WPF explicitly renders controls using DirectX, with the following benefits:

- It supports sophisticated graphics, such as arbitrary transformations, 3D rendering, multimedia, and true transparency. Skinning is supported through styles and templates.
- Its primary measurement unit is not pixel based, so applications display correctly at any DPI setting.
- It has extensive and flexible layout support, which means that you can localize an application without danger of elements overlapping.
- Its use of DirectX makes rendering fast and able to take advantage of graphics hardware acceleration.
- It offers reliable data binding.
- UIs can be described declaratively in XAML files that can be maintained independent of the "code-behind" files—this helps to separate appearance from functionality.

WPF takes some time to learn due to its size and complexity. The types for writing WPF applications are in the `System.Windows` namespace and all subnamespaces except for `System.Windows.Forms`.

Windows Forms

Windows Forms is a rich-client API that shipped with the first version of .NET Framework in 2000. Compared to WPF, Windows Forms is a relatively simple technology that provides most of the features you need in writing a typical Windows application. It also has significant relevancy in maintaining legacy applications. But compared to WPF, it has numerous drawbacks, most of which stem from it being a wrapper over GDI+ and the Win32 control library:

- Although Windows Forms provides mechanisms for DPI-awareness, it's still too easy to write applications that break on clients whose DPI settings differ from the developer's.
- The API for drawing nonstandard controls is GDI+, which, although reasonably flexible, is slow in rendering large areas (and without double buffering, might flicker).
- Controls lack true transparency.
- Most controls are noncompositional. For instance, you can't put an image control inside a tab control header. Customizing list views, combo boxes, and tab controls in a way that would be trivial with WPF is time consuming and painful in Windows Forms.
- Dynamic layout is difficult to correctly implement reliably.

The last point is an excellent reason to favor WPF over Windows Forms—even if you're writing a business application that needs just a UI and not a "user experience." The layout elements in WPF, such as `Grid`, make it easy to assemble labels and text boxes such that they always align—even after language-changing localization—without messy logic and without any flickering. Further, you don't need to bow to the lowest common denominator in screen resolution—WPF layout elements have been designed from the outset to adapt properly to resizing.

On the positive side, Windows Forms is relatively simple to learn and still has a good number of third-party controls.

The Windows Forms types are in the `System.Windows.Forms` (in *System.Windows.Forms.dll*) and `System.Drawing` (in *System.Drawing.dll*) namespaces. The latter also contains the GDI+ types for drawing custom controls.

UWP and WinUI 3

UWP is a rich-client API for writing touch-first UIs that target Windows 10+ desktop and devices. The word "Universal" refers to its ability to run on a range of Windows 10 devices, including Xbox, Surface Hub, HoloLens, and (at the time) Windows Phone.

The UWP API uses XAML and is somewhat similar to WPF. Here are its key differences:

- The primary mode of distribution for UWP apps is the Windows Store.
- UWP apps run in a sandbox to lessen the threat of malware, which means that they cannot perform tasks such as reading or writing arbitrary files, and they cannot run with administrative elevation.
- UWP relies on WinRT types that are part of the operating system (Windows), not the managed runtime. This means that when writing apps, you must nominate a Windows *version range* (such as Windows 10 build 17763 to Windows 10 build 18362). This means that you either need to target an old API or require that your customers install the latest Windows update.

Because of the limitations created by these differences, UWP never succeeded in matching the popularity of WPF and Windows Forms. To address this, Microsoft has morphed UWP into a new technology called Windows App SDK (with a UI layer called WinUI 3).

The Windows App SDK transfers the WinRT APIs from the operating system to the runtime, thereby exposing a fully managed interface and removing the necessity to target a specific operating system version range. It also does the following:

- Integrates better with the Windows Desktop APIs (Windows Forms and WPF)
- Allows you to write applications that run outside the Windows Store sandbox
- Runs atop the latest .NET (instead of being tied to .NET Core 2.2, as is the case with UWP)

Despite these improvements, WinUI 3 hasn't gained the widespread popularity of the classic Windows Desktop APIs. Windows App SDK also does not support Xbox or HoloLens at the time of writing, and requires a separate end-user download.

MAUI

MAUI (formerly Xamarin) lets you develop mobile apps in C# that target iOS and Android (as well as cross-platform desktop apps that target macOS and Windows via Catalyst and Windows App SDK).

The CLR/BCL that runs on iOS and Android is called Mono (a derivation of the open-source Mono runtime). Historically, Mono hasn't been fully compatible with .NET, and libraries that ran on both Mono and .NET would target .NET Standard. From .NET 6, however, Mono's public interface merged with .NET, making Mono, in effect, an *implementation* of .NET.

MAUI includes a unified project interface, hot reloading, and support for Blazor Desktop and hybrid apps. See *https://github.com/dotnet/maui* for more information.