O'REILLY®

Concurrency in C# Cookbook

ASYNCHRONOUS, PARALLEL, AND MULTITHREADED PROGRAMMING

Stephen Cleary

O'REILLY®

Concurrency in C# Cookbook

If you are one of the many developers uncertain about concurrent and multithreaded development, this practical cookbook will change your mind. With more than 75 code-rich recipes, author Stephen Cleary demonstrates parallel processing and asynchronous programming techniques using libraries and language features in .NET 4.5 and C# 5.0.

Concurrency is becoming more common in responsive and scalable application development, but it's been extremely difficult to code. The detailed solutions in this cookbook show you how modern tools raise the level of abstraction, making concurrency much easier than before. Complete with ready-to-use code and discussions about how and why the solution works, you get recipes for using:

- `async` and `await` for asynchronous operations
- Parallel programming with the Task Parallel Library
- The TPL Dataflow library for creating dataflow pipelines
- Capabilities that Reactive Extensions build on top of LINQ
- Unit testing with concurrent code
- Interop scenarios for combining concurrent approaches
- Immutable, threadsafe, and producer/consumer collections
- Cancellation support in your concurrent code
- Asynchronous-friendly Object-Oriented Programming
- Thread synchronization for accessing data

Stephen Cleary is a developer with extensive experience, ranging from ARM firmware to Azure. He has contributed to open source from the beginning, starting with the Boost C++ libraries, and has released several libraries and utilities of his own.

"The next big thing in computing is making massive parallelism accessible to mere mortals. Developers have more power than ever before, but expressing concurrency is still a challenge for many. Stephen turns his attention to this problem, helping us all better understand concurrency, threading, reactive programming models, parallelism, and much more in an easy-to-read but complete reference."

—**Scott Hanselman**
Principal Program Manager, ASP.NET and Azure Web Tools

"The breadth of techniques covered and the cookbook format make this the ideal reference book for modern .NET concurrency."

—**Jon Skeet**
Senior Software Engineer at Google

PROGRAMMING LANGUAGES / C#

US $39.99 **CAN $41.99**

ISBN: 978-1-449-36756-5

Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Concurrency in C# Cookbook*

"The next big thing in computing is making massive parallelism accessible to mere mortals. Developers have more power available to us than ever before, but expressing concurrency is still a challenge for many. Stephen turns his attention to this problem, helping us all better understand concurrency, threading, reactive programming models, parallelism, and much more in an easy-to-read but complete reference."

— *Scott Hanselman*
Principal Program Manager, ASP.NET and Azure Web Tools, Microsoft

"The breadth of techniques covered and the cookbook format make this the ideal reference book for modern .NET concurrency."

— *Jon Skeet*
Senior Software Engineer at Google

"Stephen Cleary has established himself as a key expert on asynchrony and parallelism in C#. This book clearly and concisely conveys the most important points and principles developers need to understand to get started and be successful with these technologies."

— *Stephen Toub*
Principal Architect, Microsoft

Concurrency in C# Cookbook

Stephen Cleary

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Concurrency in C# Cookbook

by Stephen Cleary

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian MacDonald and Rachel Roumeliotis
Production Editor: Nicole Shelby
Copyeditor: Charles Roumeliotis
Proofreader: Amanda Kersey
Indexer: Ellen Troutman
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2014: First Edition

Revision History for the First Edition:

2014-05-14: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449367565* for release details.

ISBN: 978-1-449-36756-5

[Q]

Table of Contents

Preface

I think the animal on this cover, a common palm civet, is applicable to the subject of this book. I knew nothing about this animal until I saw the cover, so I looked it up. Common palm civets are considered pests because they defecate all over ceilings and attics and make loud noises fighting with each other at the most inopportune times. Their anal scent glands emit a nauseating secretion. They have an endangered species rating of "Least Concern," which is apparently the politically correct way of saying, "Kill as many of these as you want; no one will miss them." Common palm civets enjoy eating coffee cherries, and they pass the coffee beans through. Kopi luwak, one of the most expensive coffees in the world, is made from the coffee beans extracted from civet excretions. According to the Specialty Coffee Association of America, "It just tastes bad."

This makes the common palm civet a perfect mascot for concurrent and multithreaded developement. To the uninitiated, concurrency and multithreading are undesirable. They make well-behaved code act up in the most horrendous ways. Race conditions and whatnot cause loud crashes (always, it seems, either in production or a demo). Some have gone so far as to declare "threads are evil" and avoid concurrency completely. There are a handful of developers who have developed a taste for concurrency and use it without fear; but most developers have been burned in the past by concurrency, and that experience has left a bad taste in their mouth.

However, for modern applications, concurrency is quickly becoming a requirement. Users these days expect fully responsive interfaces, and server applications are having to scale to unprecedented levels. Concurrency addresses both of these trends.

Fortunately, there are many modern libraries that make concurrency *much* easier! Parallel processing and asynchronous programming are no longer exclusively the domains of wizards. By raising the level of abstraction, these libraries make responsive and scalable application development a realistic goal for every developer. If you have been burned in the past when concurrency was extremely difficult, then I encourage you to give it another try with modern tools. We can probably never call concurrency easy, but it sure isn't as hard as it used to be!

Who Should Read This Book

This book is written for developers who want to learn modern approaches to concurrency. I do assume that you've got a fair amount of .NET experience, including an understanding of generic collections, enumerables, and LINQ. I do *not* expect that you have any multithreading or asynchronous programming knowledge. If you do have some experience in those areas, you may still find this book helpful because it introduces newer libraries that are safer and easier to use.

Concurrency is useful for any kind of application. It doesn't matter whether you work on desktop, mobile, or server applications; these days concurrency is practically a requirement across the board. You can use the recipes in this book to make user interfaces more responsive and servers more scalable. We are already at the point where concurrency is ubiquitous, and understanding these techniques and their uses is essential knowledge for the professional developer.

Why I Wrote This Book

Early in my career, I learned multithreading the hard way. After a couple of years, I learned asynchronous programming the hard way. While those were both valuable experiences, I do wish that back then I had some of the tools and resources that are available today. In particular, the `async` and `await` support in modern .NET languages is pure gold.

However, if you look around today at books and other resources for learning concurrency, they almost all start by introducing the most low-level concepts. There's excellent coverage of threads and serialization primitives, and the higher-level techniques are put off until later, if they're covered at all. I believe this is for two reasons. First, many developers of concurrency such as myself did learn the low-level concepts first, slogging through the old-school techniques. Second, many books are years old and cover now-outdated techniques; as the newer techniques have become available, these books have been updated to include them, but unfortunately placed them at the end.

I think that's backward. In fact, this book *only* covers modern approaches to concurrency. That's not to say there's no value in understanding all the low-level concepts. When I went to college for programming, I had one class where I had to build a virtual CPU from a handful of gates, and another class that covered assembly programming. In my professional career, I've never designed a CPU, and I've only written a couple dozen lines of assembly, but my understanding of the fundamentals still helps me every day. However, it's best to start with the higher-level abstractions; my first programming class was not in assembly language.

This book fills a niche: it is an introduction to (and reference for) concurrency using modern approaches. It covers several different kinds of concurreny, including parallel,

asynchronous, and reactive programming. However, it does not cover any of the old-school techniques, which are adequately covered in many other books and online resources.

Navigating This Book

This book is intended as both an introduction and as a quick reference for common solutions. The book is broken down as follows:

- Chapter 1 is an introduction to the various kinds of concurrency covered by this book: parallel, asynchronous, reactive, and dataflow.
- Chapters 2-5 are a more thorough introduction to these kinds of concurrency.
- The remaining chapters each deal with a particular aspect of concurrency, and act as a reference for solutions to common problems.

I recommend reading (or at least skimming) the first chapter, even if you're already familiar with some kinds of concurrency.

Online Resources

This book acts like a broad-spectrum introduction to several different kinds of concurrency. I've done my best to include techniques that I and others have found the most helpful, but this book is not exhaustive by any means. The following resources are the best ones I've found for a more thorough exploration of these technologies.

For parallel programming, the best resource I know of is *Parallel Programming with Microsoft .NET* by Microsoft Press, which is available online. Unfortunately, it is already a bit out of date. The section on Futures should use asynchronous code instead, and the section on Pipelines should use TPL Dataflow.

For asynchronous programming, MSDN is quite good, particularly the "Task-based Asynchronous Pattern" document.

Microsoft has also published an "Introduction to TPL Dataflow," which is the best description of TPL Dataflow.

Reactive Extensions (Rx) is a library that is gaining a lot of traction online and continues evolving. In my opinion, the best resource today for Rx is an ebook by Lee Campbell called *Introduction to Rx*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip, suggestion, or general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Pro-

fessional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/concur-c-ckbk*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This book simply would not exist without the help of so many people!

First and foremost, I'd like to acknowledge my Lord and Savior Jesus Christ. Becoming a Christian was the most important decision of my life! If you want more information on this subject, feel free to contact me via my personal web page.

Second, I thank my family for allowing me to give up so much time with them. When I started writing, I had some author friends of mine tell me, "Say goodbye to your family for the next year!" and I thought they were joking. My wife, Mandy, and our children, SD and Emma, have been very understanding while I put in long days at work followed by writing on evenings and weekends. Thank you so much. I love you!

Of course, this book would not be nearly as good as it is without my editor, Brian MacDonald, and our technical reviewers: Stephen Toub, Petr Onderka ("svick"), and Nick Paldino ("casperOne"). So if any mistakes get through, it's totally their fault. Just kidding! Their input has been invaluable in shaping (and fixing) the content, and any remaining mistakes are of course my own.

Finally, I'd like to thank some of the people I've learned these techniques from: Stephen Toub, Lucian Wischik, Thomas Levesque, Lee Campbell, the members of Stack Overflow and the MSDN Forums, and the attendees of the software conferences in and around my home state of Michigan. I appreciate being a part of the software development community, and if this book adds any value, it is only because of so many who have already shown the way. Thank you all!

CHAPTER 1

Concurrency: An Overview

Concurrency is a key aspect of beautiful software. For decades, concurrency was possible but difficult. Concurrent software was difficult to write, difficult to debug, and difficult to maintain. As a result, many developers chose the easier path and avoided concurrency. However, with the libraries and language features available for modern .NET programs, concurrency is much easier. When Visual Studio 2012 was released, Microsoft significantly lowered the bar for concurrency. Previously, concurrent programming was the domain of experts; these days, every developer can (and should) embrace concurrency.

1.1. Introduction to Concurrency

Before continuing, I'd like to clear up some terminology that I'll be using throughout this book. Let's start with *concurrency*.

Concurrency
: Doing more than one thing at a time.

I hope it's obvious how concurrency is helpful. End-user applications use concurrency to respond to user input *while* writing to a database. Server applications use concurrency to respond to a second request *while* finishing the first request. You need concurrency any time you need an application to do one thing *while* it's working on something else. Almost every software application in the world can benefit from concurrency.

At the time of this writing (2014), most developers hearing the term "concurrency" immediately think of "multithreading." I'd like to draw a distinction between these two.

Multithreading
: A form of concurrency that uses multiple threads of execution.

Multithreading literally refers to using multiple threads. As we'll see in many recipes in this book, multithreading is *one* form of concurrency, but certainly not the only one. In fact, direct use of the low-level threading types has almost no purpose in a modern

application; higher-level abstractions are more powerful and more efficient than old-school multithreading. As a consequence, I'll minimize my coverage of outdated techniques in this book. None of the multithreading recipes in this book use the `Thread` or `BackgroundWorker` types; they have been replaced with superior alternatives.

As soon as you type `new Thread()`, it's over; your project already has legacy code.

But don't get the idea that multithreading is dead! Multithreading lives on in the *thread pool*, a useful place to queue work that automatically adjusts itself according to demand. In turn, the thread pool enables another important form of concurrency: *parallel processing*.

Parallel Processing
: Doing lots of work by dividing it up among multiple threads that run concurrently.

Parallel processing (or parallel programming) uses multithreading to maximize the use of multiple processors. Modern CPUs have multiple cores, and if there's a lot of work to do, then it makes no sense to just make one core do all the work while the others sit idle. Parallel processing will split up the work among multiple threads, which can each run independently on a different core.

Parallel processing is one type of multithreading, and multithreading is one type of concurrency. There's another type of concurrency that is important in modern applications but is not (currently) familiar to many developers: *asynchronous programming*.

Asynchronous Programming
: A form of concurrency that uses futures or callbacks to avoid unnecessary threads.

A *future* (or *promise*) is a type that represents some operation that will complete in the future. The modern future types in .NET are `Task` and `Task<TResult>`. Older asynchronous APIs use callbacks or events instead of futures. Asynchronous programming is centered around the idea of an *asynchronous operation*: some operation that is started that will complete some time later. While the operation is in progress, it does not block the original thread; the thread that starts the operation is free to do other work. When the operation completes, it notifies its future or invokes its completion callback event to let the application know the operation is finished.

Asynchronous programming is a powerful form of concurrency, but until recently, it required extremely complex code. The `async` and `await` support in VS2012 make asynchronous programming almost as easy as synchronous (nonconcurrent) programming.

Another form of concurrency is *reactive programming*. Asynchronous programming implies that the application will start an operation that will complete once at a later time. Reactive programming is closely related to asynchronous programming, but is built on *asynchronous events* instead of *asynchronous operations*. Asynchronous events may not have an actual "start," may happen at any time, and may be raised multiple times. One example is user input.

Reactive Programming
: A declarative style of programming where the application reacts to events.

If you consider an application to be a massive state machine, the application's behavior can be described as reacting to a series of events by updating its state at each event. This is not as abstract or theoretical as it sounds; modern frameworks make this approach quite useful in real-world applications. Reactive programming is not necessarily concurrent, but it is closely related to concurrency, so we'll be covering the basics in this book.

Usually, a mixture of techniques are used in a concurrent program. Most applications at least use multithreading (via the thread pool) and asynchronous programming. Feel free to mix and match all the various forms of concurrency, using the appropriate tool for each part of the application.

1.2. Introduction to Asynchronous Programming

Asynchronous programming has two primary benefits. The first benefit is for end-user GUI programs: asynchronous programming enables responsiveness. We've all used a program that temporarily locks up while it's working; an asynchronous program can remain responsive to user input while it's working. The second benefit is for server-side programs: asynchronous programming enables scalability. A server application can scale somewhat just by using the thread pool, but an asynchronous server application can usually scale an order of magnitude better than that.

Modern asynchronous .NET applications use two keywords: `async` and `await`. The `async` keyword is added to a method declaration, and its primary purpose is to enable the `await` keyword within that method (the keywords were introduced as a pair for backward-compatibility reasons). An `async` method should return `Task<T>` if it returns a value, or `Task` if it does not return a value. These task types represent futures; they notify the calling code when the `async` method completes.

Avoid `async void`! It is possible to have an `async` method return `void`, but you should only do this if you're writing an `async` event handler. A regular `async` method without a return value should return `Task`, not `void`.

With that background, let's take a quick look at an example:

```
async Task DoSomethingAsync()
{
  int val = 13;

  // Asynchronously wait 1 second.
  await Task.Delay(TimeSpan.FromSeconds(1));

  val *= 2;

  // Asynchronously wait 1 second.
  await Task.Delay(TimeSpan.FromSeconds(1));

  Trace.WriteLine(val);
}
```

An `async` method begins executing synchronously, just like any other method. Within an `async` method, the `await` keyword performs an *asynchronous wait* on its argument. First, it checks whether the operation is already complete; if it is, it continues executing (synchronously). Otherwise, it will pause the `async` method and return an incomplete task. When that operation completes some time later, the `async` method will resume executing.

You can think of an `async` method as having several synchronous portions, broken up by `await` statements. The first synchronous portion executes on whetever thread calls the method, but where do the other synchronous portions execute? The answer is a bit complicated.

When you `await` a task (the most common scenario), a *context* is captured when the `await` decides to pause the method. This context is the current `SynchronizationContext` unless it is `null`, in which case the context is the current `TaskScheduler`. The method resumes executing within that captured context. Usually, this context is the UI context (if you're on the UI thread), an ASP.NET request context (if you're processing an ASP.NET request), or the thread pool context (most other situations).

So, in the preceding code, all the synchronous portions will attempt to resume on the original context. If you call `DoSomethingAsync` from a UI thread, each of its synchronous portions will run on that UI thread; but if you call it from a thread-pool thread, each of its synchronous portions will run on a thread-pool thread.

You can avoid this default behavior by awaiting the result of the `ConfigureAwait` extension method and passing `false` for the `continueOnCapturedContext` parameter. The following code will start on the calling thread, and after it is paused by an `await`, it will resume on a thread-pool thread:

```
async Task DoSomethingAsync()
{
  int val = 13;
```

```
  // Asynchronously wait 1 second.
  await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

  val *= 2;

  // Asynchronously wait 1 second.
  await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

  Trace.WriteLine(val.ToString());
}
```

It's good practice to always call `ConfigureAwait` in your core "library" methods, and only resume the context when you need it—in your outer "user interface" methods.

The `await` keyword is not limited to working with tasks; it can work with any kind of *awaitable* that follows a certain pattern. As one example, the Windows Runtime API defines its own interfaces for asynchronous operations. These are not convertible to `Task`, but they do follow the awaitable pattern, so you can directly `await` them. These awaitables are more common in Windows Store applications, but most of the time `await` will take a `Task` or `Task<T>`.

There are two basic ways to create a `Task` instance. Some tasks represent actual code that a CPU has to execute; these computational tasks should be created by calling `Task.Run` (or `TaskFactory.StartNew` if you need them to run on a particular scheduler). Other tasks represent a *notification*; these event-based tasks are created by `TaskCompletionSource<T>` (or one of its shortcuts). Most I/O tasks use `TaskCompletionSource<T>`.

Error handling is natural with `async` and `await`. In the following code snippet, `PossibleExceptionAsync` may throw a `NotSupportedException`, but `TrySomethingAsync` can catch the exception naturally. The caught exception has its stack trace properly preserved and is not artificailly wrapped in a `TargetInvocationException` or `AggregateException`:

```
async Task TrySomethingAsync()
{
  try
  {
    await PossibleExceptionAsync();
  }
  catch (NotSupportedException ex)
  {
    LogException(ex);
    throw;
```

```
  }
}
```

When an `async` method throws (or propagates) an exception, the exception is placed on its returned `Task` and the `Task` is completed. When that `Task` is awaited, the `await` operator will retrieve that exception and (re)throw it in a way such that its original stack trace is preserved. Thus, code like this would work as expected if `PossibleExceptionAsync` was an `async` method:

```
async Task TrySomethingAsync()
{
  // The exception will end up on the Task, not thrown directly.
  Task task = PossibleExceptionAsync();

  try
  {
    // The Task's exception will be raised here, at the await.
    await task;
  }
  catch (NotSupportedException ex)
  {
    LogException(ex);
    throw;
  }
}
```

There's one other important guideline when it comes to `async` methods: once you start using `async`, it's best to allow it to grow through your code. If you call an `async` method, you should (eventually) `await` the task it returns. Resist the temptation of calling `Task.Wait` or `Task<T>.Result`; this could cause a deadlock. Consider this method:

```
async Task WaitAsync()
{
  // This await will capture the current context ...
  await Task.Delay(TimeSpan.FromSeconds(1));
  // ... and will attempt to resume the method here in that context.
}

void Deadlock()
{
  // Start the delay.
  Task task = WaitAsync();

  // Synchronously block, waiting for the async method to complete.
  task.Wait();
}
```

This code will deadlock if called from a UI or ASP.NET context. This is because both of those contexts only allow one thread in at a time. `Deadlock` will call `WaitAsync`, which begins the delay. `Deadlock` then (synchronously) waits for that method to complete, blocking the context thread. When the delay completes, `await` attempts to resume

`WaitAsync` within the captured context, but it cannot because there is already a thread blocked in the context, and the context only allows one thread at a time. Deadlock can be prevented two ways: you can use `ConfigureAwait(false)` within `WaitAsync` (which causes `await` to ignore its context), or you can `await` the call to `WaitAsync` (making `Deadlock` into an `async` method).

If you use `async`, it's best to use `async` all the way.

If you would like a more complete introduction to `async`, *Async in C# 5.0* by Alex Davies (O'Reilly) is an excellent resource. Also, the online documentation that Microsoft has provided for `async` is better than usual; I recommend reading at least the the `async` overview and the Task-based Asynchronous Pattern (TAP) overview. If you really want to go deep, there's an official FAQ and blog that have tremendous amounts of information.

1.3. Introduction to Parallel Programming

Parallel programming should be used any time you have a fair amount of computation work that can be split up into independent chunks of work. Parallel programming increases the CPU usage temporarily to improve throughput; this is desirable on client systems where CPUs are often idle but is usually not appropriate for server systems. Most servers have some parallelism built in; for example, ASP.NET will handle multiple requests in parallel. Writing parallel code on the server may still be useful in some situations (if you *know* that the number of concurrent users will always be low), but in general, parallel programming on the server would work against the built-in parallelism and would not provide any real benefit.

There are two forms of parallelism: *data parallelism* and *task parallelism*. Data parallelism is when you have a bunch of data items to process, and the processing of each piece of data is mostly independent from the other pieces. Task parallelism is when you have a pool of work to do, and each piece of work is mostly independent from the other pieces. Task parallelism may be dynamic; if one piece of work results in several additional pieces of work, they can be added to the pool of work.

There are a few different ways to do data parallelism. `Parallel.ForEach` is similar to a `foreach` loop and should be used when possible. `Parallel.ForEach` is covered in Recipe 3.1. The `Parallel` class also supports `Parallel.For`, which is similar to a `for` loop and can be used if the data processing depends on the index. Code using `Parallel.ForEach` looks like this:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

Another option is PLINQ (Parallel LINQ), which provides an `AsParallel` extension method for LINQ queries. `Parallel` is more resource friendly than PLINQ; `Parallel` will play more nicely with other processes in the system, while PLINQ will (by default) attempt to spread itself over all CPUs. The downside to `Parallel` is that it is more explicit; PLINQ has more elegant code in many cases. PLINQ is covered in Recipe 3.5:

```
IEnumerable<bool> PrimalityTest(IEnumerable<int> values)
{
    return values.AsParallel().Select(val => IsPrime(val));
}
```

Regardless of the method you choose, one guideline stands out when doing parallel processing.

The chunks of work should be as independent from each other as possible.

As long as your chunk of work is independent from all other chunks, you maximize your parallelism. As soon as you start sharing state between multiple threads, you have to synchronize access to that shared state, and your application becomes less parallel. We'll cover synchronization in more detail in Chapter 11.

The output of your parallel processing can be handled various ways. You can place the results in some kind of a concurrent collection, or you can aggregate the results into a summary. Aggregation is common in parallel processing; this kind of map/reduce functionality is also supported by the `Parallel` class method overloads. We'll look at aggregation in more detail in Recipe 3.2.

Now let's turn to task parallelism. Data parallelism is focused on processing data; task parallelism is just about doing work.

One `Parallel` method that does a type of fork/join task parallelism is `Parallel.Invoke`. This is covered in Recipe 3.3; you just pass in the delegates you want to execute in parallel:

```
void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
```

```
    );
}

void ProcessPartialArray(double[] array, int begin, int end)
{
    // CPU-intensive processing...
}
```

The `Task` type was originally introduced for task parallelism, though these days it's also used for asynchronous programming. A `Task` instance—as used in task parallelism—represents some work. You can use the `Wait` method to wait for a task to complete, and you can use the `Result` and `Exception` properties to retrieve the results of that work. Code using `Task` directly is more complex than code using `Parallel`, but it can be useful if you don't know the structure of the parallelism until runtime. With this kind of dynamic parallelism, you don't know how many pieces of work you need to do at the beginning of the processing; you find it out as you go along. Generally, a dynamic piece of work should start whatever child tasks it needs and then wait for them to complete. The `Task` type has a special flag, `TaskCreationOptions.AttachedToParent`, which you could use for this. Dynamic parallelism is covered in Recipe 3.4.

Task parallelism should strive to be independent, just like data parallelism. The more independent your delegates can be, the more efficient your program can be. With task parallelism, be especially careful of variables captured in closures. Remember that closures capture references (not values), so you can end up with sharing that isn't obvious.

Error handling is similar for all kinds of parallelism. Since operations are proceeding in parallel, it is possible for multiple exceptions to occur, so they are wrapped up in an `AggregateException`, which is thrown to your code. This behavior is consistent across `Parallel.ForEach`, `Parallel.Invoke`, `Task.Wait`, etc. The `AggregateException` type has some useful `Flatten` and `Handle` methods to simplify the error handling code:

```
try
{
    Parallel.Invoke(() => { throw new Exception(); },
        () => { throw new Exception(); });
}
catch (AggregateException ex)
{
    ex.Handle(exception =>
    {
        Trace.WriteLine(exception);
        return true; // "handled"
    });
}
```

Usually, you don't have to worry about how the work is handled by the thread pool. Data and task parallelism use dynamically adjusting partitioners to divide work among worker threads. The thread pool increases its thread count as necessary. Thread-pool

threads use work-stealing queues. Microsoft put a lot of work into making each part as efficient as possible, and there are a large number of knobs you can tweak if you need maximum performance. As long as your tasks are not extremely short, they should work well with the default settings.

Tasks should not be extremely short, nor extremely long.

If your tasks are too short, then the overhead of breaking up the data into tasks and scheduling those tasks on the thread pool becomes significant. If your tasks are too long, then the thread pool cannot dynamically adjust its work balancing efficiently. It's difficult to determine how short is too short and how long is too long; it really depends on the problem being solved and the approximate capabilities of the hardware. As a general rule, I try to make my tasks as short as possible without running into performance issues (you'll see your performance suddenly degrade when your tasks are too short). Even better, instead of using tasks directly, use the `Parallel` type or PLINQ. These higher-level forms of parallelism have partitioning built in to handle this automatically for you (and adjust as necessary at runtime).

If you want to dive deeper into parallel programming, the best book on the subject is *Parallel Programming with Microsoft .NET*, by Colin Campbell et al. (MSPress).

1.4. Introduction to Reactive Programming (Rx)

Reactive programming has a higher learning curve than other forms of concurrency, and the code can be harder to maintain unless you keep up with your reactive skills. If you're willing to learn it, though, reactive programming is extremely powerful. Reactive programming allows you to treat a stream of events like a stream of data. As a rule of thumb, if you use any of the event arguments passed to an event, then your code would benefit from using Rx instead of a regular event handler.

Reactive programming is based around the notion of observable streams. When you subscribe to an observable stream, you'll receive any number of data items (`OnNext`) and then the stream may end with a single error (`OnError`) or "end of stream" notification (`OnCompleted`). Some observable streams never end. The actual interfaces look like this:

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

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

However, you should never implement these interfaces. The Reactive Extensions (Rx) library by Microsoft has all the implementations you should ever need. Reactive code ends up looking very much like LINQ; you can think of it as "LINQ to events." The following code starts with some unfamiliar operators (`Interval` and `Timestamp`) and ends with a `Subscribe`, but in the middle are some operators that should be familiar from LINQ: `Where` and `Select`. Rx has everything that LINQ does and adds in a large number of its own operators, particularly ones that deal with time:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x));
```

The example code starts with a counter running off a periodic timer (`Interval`) and adds a timestamp to each event (`Timestamp`). It then filters the events to only include even counter values (`Where`), selects the timestamp values (`Timestamp`), and then as each resulting timestamp value arrives, writes it to the debugger (`Subscribe`). Don't worry if you don't understand the new operators, such as `Interval`: we'll cover those later. For now, just keep in mind that this is a LINQ query very similar to the ones with which you are already familiar. The main difference is that LINQ to Objects and LINQ to Entities use a *"pull" model*, where the enumeration of a LINQ query pulls the data through the query, while LINQ to events (Rx) uses a *"push" model*, where the events arrive and travel through the query by themselves.

The definition of an observable stream is independent from its subscriptions. The last example is the same as this one:

```
IObservable<DateTimeOffset> timestamps =
    Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp);
timestamps.Subscribe(x => Trace.WriteLine(x));
```

It is normal for a type to define the observable streams and make them available as an `IObservable<T>` resource. Other types can then subscribe to those streams or combine them with other operators to create another observable stream.

An Rx subscription is also a resource. The `Subscribe` operators return an `IDisposable` that represents the subscription. When you are done responding to that observable stream, dispose of the subscription.

Subscriptions behave differently with hot and cold observables. A *hot observable* is a stream of events that is always going on, and if there are no subscribers when the events come in, they are lost. For example, mouse movement is a hot observable. A *cold observable* is an observable that doesn't have incoming events all the time. A cold observable will react to a subscription by starting the sequence of events. For example, an HTTP download is a cold observable; the subscription causes the HTTP request to be sent.

The `Subscribe` operator should always take an error handling parameter as well. The preceding examples do not; the following is a better example that will respond appropriately if the observable stream ends in an error:

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Where(x => x.Value % 2 == 0)
    .Select(x => x.Timestamp)
    .Subscribe(x => Trace.WriteLine(x),
        ex => Trace.WriteLine(ex));
```

One type that is useful when experimenting with Rx is `Subject<T>`. This "subject" is like a manual implementation of an observable stream. Your code can call `OnNext`, `OnError`, and `OnCompleted`, and the subject will forward those calls to its subscribers. `Subject<T>` is great for experimenting, but in production code, you should use operators like those covered in Chapter 5.

There are tons of useful Rx operators, and I only cover a few selected ones in this book. For more information on Rx, I recommend the excellent online book Introduction to Rx.

1.5. Introduction to Dataflows

TPL Dataflow is an interesting mix of asynchronous and parallel technologies. It is useful when you have a sequence of processes that need to be applied to your data. For example, you may need to download data from a URL, parse it, and then process it in parallel with other data. TPL Dataflow is commonly used as a simple pipeline, where data enters one end and travels until it comes out the other. However, TPL Dataflow is far more powerful than this; it is capable of handling any kind of mesh. You can define forks, joins, and loops in a mesh, and TPL Dataflow will handle them appropriately. Most of the time, though, TPL Dataflow meshes are used as a pipeline.

The basic building unit of a dataflow mesh is a *dataflow block*. A block can either be a target block (receiving data), a source block (producing data), or both. Source blocks can be linked to target blocks to create the mesh; linking is covered in Recipe 4.1. Blocks are semi-independent; they will attempt to process data as it arrives and push the results downstream. The usual way of using TPL Dataflow is to create all the blocks, link them together, and then start putting data in one end. The data then comes out of the other end by itself. Again, Dataflow is more powerful than this; it is possible to break links

and create new blocks and add them to the mesh *while* there is data flowing through it, but this is a very advanced scenario.

Target blocks have buffers for the data they receive. This allows them to accept new data items even if they are not ready to process them yet, keeping data flowing through the mesh. This buffering can cause problems in fork scenarios, where one source block is linked to two target blocks. When the source block has data to send downstream, it starts offering it to its linked blocks one at a time. By default, the first target block would just take the data and buffer it, and the second target block would never get any. The fix for this situation is to limit the target block buffers by making them nongreedy; we cover this in Recipe 4.4.

A block will fault when something goes wrong, for example, if the processing delegate throws an exception when processing a data item. When a block faults, it will stop receiving data. By default, it will not take down the whole mesh; this gives you the capability to rebuild that part of the mesh or redirect the data. However, this is an advanced scenario; most times, you want the faults to propagate along the links to the target blocks. Dataflow supports this option as well; the only tricky part is that when an exception is propagated along a link, it is wrapped in an `AggregateException`. So, if you have a long pipeline, you could end up with a deeply nested exception; the `AggregateException.Flatten` method can be used to work around this:

```
try
{
    var multiplyBlock = new TransformBlock<int, int>(item =>
    {
        if (item == 1)
            throw new InvalidOperationException("Blech.");
        return item * 2;
    });
    var subtractBlock = new TransformBlock<int, int>(item => item - 2);
    multiplyBlock.LinkTo(subtractBlock,
        new DataflowLinkOptions { PropagateCompletion = true });

    multiplyBlock.Post(1);
    subtractBlock.Completion.Wait();
}
catch (AggregateException exception)
{
    AggregateException ex = exception.Flatten();
    Trace.WriteLine(ex.InnerException);
}
```

Dataflow error handling is covered in more detail in Recipe 4.2.

At first glance, dataflow meshes sound very much like observable streams, and they do have much in common. Both meshes and streams have the concept of data items passing through them. Also, both meshes and streams have the notion of a normal completion (a notification that no more data is coming), as well as a faulting completion (a notifi-

cation that some error occurred during data processing). However, Rx and TPL Dataflow do not have the same capabilities. Rx observables are generally better than dataflow blocks when doing anything related to timing. Dataflow blocks are generally better than Rx observables when doing parallel processing. Conceptually, Rx works more like setting up callbacks: each step in the observable directly calls the next step. In contrast, each block in a dataflow mesh is very independent from all the other blocks. Both Rx and TPL Dataflow have their own uses, with some amount of overlap. However, they also work quite well together; we'll cover Rx and TPL Dataflow interoperability in Recipe 7.7.

The most common block types are `TransformBlock<TInput, TOutput>` (similar to LINQ's `Select`), `TransformManyBlock<TInput, TOutput>` (similar to LINQ's `SelectMany`), and `ActionBlock<T>`, which executes a delegate for each data item. For more information on TPL Dataflow, I recommend the MSDN documentation and the "Guide to Implementing Custom TPL Dataflow Blocks."

1.6. Introduction to Multithreaded Programming

A *thread* is an independent executor. Each process has multiple threads in it, and each of those threads can be doing different things simultaneously. Each thread has its own independent stack but shares the same memory with all the other threads in a process. In some applications, there is one thread that is special. User interface applications have a single UI thread; Console applications have a single main thread.

Every .NET application has a thread pool. The thread pool maintains a number of worker threads that are waiting to execute whatever work you have for them to do. The thread pool is responsible for determining how many threads are in the thread pool at any time. There are dozens of configuration settings you can play with to modify this behavior, but I recommend that you leave it alone; the thread pool has been carefully tuned to cover the vast majority of real-world scenarios.

There is almost no need to ever create a new thread yourself. The only time you should ever create a `Thread` instance is if you need an STA thread for COM interop.

A thread is a low-level abstraction. The thread pool is a slightly higher level of abstraction; when code queues work to the thread pool, it will take care of creating a thread if necessary. The abstractions covered in this book are higher still: parallel and dataflow processing queues work to the thread pool as necessary. Code using these higher abstractions is easier to get right.

For this reason, the `Thread` and `BackgroundWorker` types are not covered at all in this book. They have had their time, and that time is over.

1.7. Collections for Concurrent Applications

There are a couple of collection categories that are useful for concurrent programming: concurrent collections and immutable collections. Both of these collection categories are covered in Chapter 8. Concurrent collections allow multiple threads to update them simulatenously in a safe way. Most concurrent collections use *snapshots* to allow one thread to enumerate the values while another thread may be adding or removing values. Concurrent collections are usually more efficient than just protecting a regular collection with a lock.

Immutable collections are a bit different. An immutable collection cannot actually be modified; instead, to modify an immutable collection, you create a new collection that represents the modified collection. This sounds horribly inefficient, but immutable collections share as much memory as possible between collection instances, so it's not as bad as it sounds. The nice thing about immutable collections is that all operations are pure, so they work very well with functional code.

1.8. Modern Design

Most concurrent technologies have one similar aspect: they are functional in nature. I don't mean *functional* as in "they get the job done," but rather *functional* as a style of programming that is based on function composition. If you adopt a functional mindset, your concurrent designs will be less convoluted.

One principle of functional programming is purity (that is, avoiding side effects). Each piece of the solution takes some value(s) as input and produces some value(s) as output. As much as possible, you should avoid having these pieces depend on global (or shared) variables or update global (or shared) data structures. This is true whether the piece is an `async` method, a parallel task, an Rx operation, or a dataflow block. Of course, sooner or later your computations will have to have an effect, but you'll find your code is cleaner if you can handle the *processing* with pure pieces and then perform updates with the *results*.

Another principle of functional programming is immutability. Immutability means that a piece of data cannot change. One reason that immutable data is useful for concurrent programs is that you never need synchronization for immutable data; the fact that it cannot change makes synchronization unnecessary. Immutable data also helps you avoid side effects. As of this writing (2014), there isn't much adoption of immutable data, but this book has several receipes covering immutable data structures.

1.9. Summary of Key Technologies

The .NET framework has had some support for asynchronous programming since the very beginning. However, asynchronous programming was difficult until 2012,

when .NET 4.5 (along with C# 5.0 and VB 2012) introduced the `async` and `await` keywords. This book will use the modern `async`/`await` approach for all asynchronous recipes, and we also have some recipes showing how to interoperate between `async` and the older asynchronous programming patterns. If you need support for older platforms, get the `Microsoft.Bcl.Async` NuGet package.

Do not use `Microsoft.Bcl.Async` to enable `async` code on ASP.NET running on .NET 4.0! The ASP.NET pipeline was updated in .NET 4.5 to be `async`-aware, and you must use .NET 4.5 or newer for `async` ASP.NET projects.

The Task Parallel Library was introduced in .NET 4.0 with full support for both data and task parallelism. However, it is not normally available on platforms with fewer resources, such as mobile phones. The TPL is built in to the .NET framework.

The Reactive Extensions team has worked hard to support as many platforms as possible. Reactive Extensions, like `async` and `await`, provide benefits for all sorts of applications, both client and server. Rx is available in the `Rx-Main` NuGet package.

The TPL Dataflow library only supports newer platforms. TPL Dataflow is officially distributed in the `Microsoft.Tpl.Dataflow` NuGet package.

Concurrent collections are part of the full .NET framework, while immutable collections are available in the `Microsoft.Bcl.Immutable` NuGet package. Table 1-1 summarizes the support of key platforms for different techniques.

Table 1-1. Platform support for concurrency

Platform	async	Parallel	Rx	Dataflow	Concurrent collections	Immutable collections
.NET 4.5	✓	✓	✓	✓	✓	✓
.NET 4.0	✓	✓	✓	✗	✓	✗
Mono iOS/Droid	✓	✓	✓	✓	✓	✓
Windows Store	✓	✓	✓	✓	✓	✓
Windows Phone Apps 8.1	✓	✓	✓	✓	✓	✓
Windows Phone SL 8.0	✓	✗	✓	✓	✗	✓
Windows Phone SL 7.1	✓	✗	✓	✗	✗	✗
Silverlight 5	✓	✗	✓	✗	✗	✗

CHAPTER 2
Async Basics

This chapter introduces you to the basics of using `async` and `await` for asynchronous operations. This chapter only deals with naturally asynchronous operations, which are operations such as HTTP requests, database commands, and web service calls.

If you have a CPU-intensive operation that you want to treat as though it were asynchronous (e.g., so it doesn't block the UI thread), then see Chapter 3 and Recipe 7.4. Also, this chapter only deals with operations that are started once and complete once; if you need to handle streams of events, then see Chapter 5.

To use `async` on older platforms, install the NuGet package `Microsoft.Bcl.Async` into your application. Some platforms support `async` natively, and some should have the package installed (see Table 2-1):

Table 2-1. Platform support for `async`

Platform	Dataflow support
.NET 4.5	✓
.NET 4.0	NuGet
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone 7.1	NuGet
Silverlight 5	NuGet

2.1. Pausing for a Period of Time

Problem

You need to (asynchronously) wait for a period of time. This can be useful when unit testing or implementing retry delays. This solution can also be useful for simple timeouts.

Solution

The `Task` type has a static method `Delay` that returns a task that completes after the specified time.

If you are using the `Microsoft.Bcl.Async` NuGet library, the `Delay` member is on the `TaskEx` type, not the `Task` type.

This example defines a task that completes asynchronously, for use with unit testing. When faking an asynchronous operation, it's important to test at least synchronous success and asynchronous success as well as asynchronous failure. This example returns a task used for the asynchronous success case:

```
static async Task<T> DelayResult<T>(T result, TimeSpan delay)
{
    await Task.Delay(delay);
    return result;
}
```

This next example is a simple implementation of an exponential backoff, that is, a retry strategy where you increase the delays between retries. Exponential backoff is a best practice when working with web services to ensure the server does not get flooded with retries.

For production code, I would recommend a more thorough solution, such as the Transient Error Handling Block in Microsoft's Enterprise Library; the following code is just a simple example of `Task.Delay` usage.

```
static async Task<string> DownloadStringWithRetries(string uri)
{
    using (var client = new HttpClient())
    {
        // Retry after 1 second, then after 2 seconds, then 4.
```

```
        var nextDelay = TimeSpan.FromSeconds(1);
        for (int i = 0; i != 3; ++i)
        {
            try
            {
                return await client.GetStringAsync(uri);
            }
            catch
            {
            }

            await Task.Delay(nextDelay);
            nextDelay = nextDelay + nextDelay;
        }

        // Try one last time, allowing the error to propogate.
        return await client.GetStringAsync(uri);
    }
}
```

This final example uses `Task.Delay` as a simple timeout; in this case, the desired semantics are to return `null` if the service does not respond within three seconds:

```
static async Task<string> DownloadStringWithTimeout(string uri)
{
    using (var client = new HttpClient())
    {
        var downloadTask = client.GetStringAsync(uri);
        var timeoutTask = Task.Delay(3000);

        var completedTask = await Task.WhenAny(downloadTask, timeoutTask);
        if (completedTask == timeoutTask)
            return null;
        return await downloadTask;
    }
}
```

Discussion

`Task.Delay` is a fine option for unit testing asynchronous code or for implementing retry logic. However, if you need to implement a timeout, a `CancellationToken` is usually a better choice.

See Also

Recipe 2.5 covers how `Task.WhenAny` is used to determine which task completes first. Recipe 9.3 covers using `CancellationToken` as a timeout.

2.2. Returning Completed Tasks

Problem

You need to implement a synchronous method with an asynchronous signature. This situation can arise if you are inheriting from an asynchronous interface or base class but wish to implement it synchronously. This technique is particularly useful when unit testing asynchronous code, when you need a simple stub or mock for an asynchronous interface.

Solution

You can use `Task.FromResult` to create and return a new `Task<T>` that is already completed with the specified value:

```
interface IMyAsyncInterface
{
    Task<int> GetValueAsync();
}

class MySynchronousImplementation : IMyAsyncInterface
{
    public Task<int> GetValueAsync()
    {
        return Task.FromResult(13);
    }
}
```

If you're using `Microsoft.Bcl.Async`, the `FromResult` method is on the `TaskEx` type.

Discussion

If you are implementing an asynchronous interface with synchronous code, avoid any form of blocking. It is not natural for an asynchronous method to block and then return a completed task. For a counterexample, consider the `Console` text readers in .NET 4.5. `Console.In.ReadLineAsync` will actually block the calling thread until a line is read, and then will return a completed task. This behavior is not intuitive and has surprised many developers. If an asynchronous method blocks, it prevents the calling thread from starting other tasks, which interferes with concurrency and may even cause a deadlock.

`Task.FromResult` provides synchronous tasks only for successful results. If you need a task with a different kind of result (e.g., a task that is completed with a

NotImplementedException), then you can create your own helper method using TaskCompletionSource:

```
static Task<T> NotImplementedAsync<T>()
{
    var tcs = new TaskCompletionSource<T>();
    tcs.SetException(new NotImplementedException());
    return tcs.Task;
}
```

Conceptually, Task.FromResult is just a shorthand for TaskCompletionSource, very similar to the preceding code.

If you regularly use Task.FromResult with the same value, consider caching the actual task. For example, if you create a Task<int> with a zero result once, then you avoid creating extra instances that will have to be garbage-collected:

```
private static readonly Task<int> zeroTask = Task.FromResult(0);
static Task<int> GetValueAsync()
{
    return zeroTask;
}
```

See Also

Recipe 6.1 covers unit testing asynchronous methods.

Recipe 10.1 covers inheritance of async methods.

2.3. Reporting Progress

Problem

You need to respond to progress while an asynchronous operation is executing.

Solution

Use the provided IProgress<T> and Progress<T> types. Your async method should take an IProgress<T> argument; the T is whatever type of progress you need to report:

```
static async Task MyMethodAsync(IProgress<double> progress = null)
{
    double percentComplete = 0;
    while (!done)
    {
        ...
        if (progress != null)
            progress.Report(percentComplete);
    }
}
```

Calling code can use it as such:

```
static async Task CallMyMethodAsync()
{
    var progress = new Progress<double>();
    progress.ProgressChanged += (sender, args) =>
    {
        ...
    };
    await MyMethodAsync(progress);
}
```

Discussion

By convention, the `IProgress<T>` parameter may be `null` if the caller does not need progress reports, so be sure to check for this in your `async` method.

Bear in mind that the `IProgress<T>.Report` method may be asynchronous. This means that `MyMethodAsync` may continue executing before the progress is actually reported. For this reason, it's best to define `T` as an *immutable type* or at least a value type. If `T` is a mutable reference type, then you'll have to create a separate copy yourself each time you call `IProgress<T>.Report`.

`Progress<T>` will capture the current context when it is constructed and will invoke its callback within that context. This means that if you construct the `Progress<T>` on the UI thread, then you can update the UI from its callback, even if the asynchronous method is invoking `Report` from a background thread.

When a method supports progress reporting, it should also make a best effort to support cancellation.

See Also

Recipe 9.4 covers how to support cancellation in an asynchronous method.

2.4. Waiting for a Set of Tasks to Complete

Problem

You have several tasks and need to wait for them all to complete.

Solution

The framework provides a `Task.WhenAll` method for this purpose. This method takes several tasks and returns a task that completes when all of those tasks have completed:

```
Task task1 = Task.Delay(TimeSpan.FromSeconds(1));
Task task2 = Task.Delay(TimeSpan.FromSeconds(2));
Task task3 = Task.Delay(TimeSpan.FromSeconds(1));

await Task.WhenAll(task1, task2, task3);
```

If all the tasks have the same result type and they all complete successfully, then the `Task.WhenAll` task will return an array containing all the task results:

```
Task task1 = Task.FromResult(3);
Task task2 = Task.FromResult(5);
Task task3 = Task.FromResult(7);

int[] results = await Task.WhenAll(task1, task2, task3);

// "results" contains { 3, 5, 7 }
```

There is an overload of `Task.WhenAll` that takes an `IEnumerable` of tasks; however, I do not recommend that you use it. Whenever I mix asynchronous code with LINQ, I find the code is clearer when I explicitly "reify" the sequence (i.e., evaluate the sequence, creating a collection):

```
static async Task<string> DownloadAllAsync(IEnumerable<string> urls)
{
    var httpClient = new HttpClient();

    // Define what we're going to do for each URL.
    var downloads = urls.Select(url => httpClient.GetStringAsync(url));
    // Note that no tasks have actually started yet
    //  because the sequence is not evaluated.

    // Start all URLs downloading simultaneously.
    Task<string>[] downloadTasks = downloads.ToArray();
    // Now the tasks have all started.

    // Asynchronously wait for all downloads to complete.
    string[] htmlPages = await Task.WhenAll(downloadTasks);

    return string.Concat(htmlPages);
}
```

If you are using the `Microsoft.Bcl.Async` NuGet library, the `WhenAll` member is on the `TaskEx` type, not the `Task` type.

Discussion

If any of the tasks throws an exception, then `Task.WhenAll` will fault its returned task with that exception. If multiple tasks throw an exception, then all of those exceptions are placed on the `Task` returned by `Task.WhenAll`. However, when that task is awaited, only one of them will be thrown. If you need each specific exception, you can examine the `Exception` property on the `Task` returned by `Task.WhenAll`:

```
static async Task ThrowNotImplementedExceptionAsync()
{
    throw new NotImplementedException();
}

static async Task ThrowInvalidOperationExceptionAsync()
{
    throw new InvalidOperationException();
}

static async Task ObserveOneExceptionAsync()
{
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    try
    {
        await Task.WhenAll(task1, task2);
    }
    catch (Exception ex)
    {
        // "ex" is either NotImplementedException or InvalidOperationException.
        ...
    }
}

static async Task ObserveAllExceptionsAsync()
{
    var task1 = ThrowNotImplementedExceptionAsync();
    var task2 = ThrowInvalidOperationExceptionAsync();

    Task allTasks = Task.WhenAll(task1, task2);
    try
    {
        await allTasks;
    }
    catch
    {
        AggregateException allExceptions = allTasks.Exception;
        ...
    }
}
```

Most of the time, I do *not* observe all the exceptions when using `Task.WhenAll`. It is usually sufficient to just respond to the first error that was thrown, rather than all of them.

See Also

Recipe 2.5 covers a way to wait for *any* of a collection of tasks to complete.

Recipe 2.6 covers waiting for a collection of tasks to complete and performing actions as each one completes.

Recipe 2.8 covers exception handling for `async Task` methods.

2.5. Waiting for Any Task to Complete

Problem

You have several tasks and need to respond to just one of them completing. The most common situation for this is when you have multiple independent attempts at an operation, with a first-one-takes-all kind of structure. For example, you could request stock quotes from multiple web services simultaneously, but you only care about the first one that responds.

Solution

Use the `Task.WhenAny` method. This method takes a sequence of tasks and returns a task that completes when any of the tasks complete. The result of the returned task is the task that completed. Don't worry if that sounds confusing; it's one of those things that's difficult to explain but easy to demonstrate:

```
// Returns the length of data at the first URL to respond.
private static async Task<int> FirstRespondingUrlAsync(string urlA, string urlB)
{
    var httpClient = new HttpClient();

    // Start both downloads concurrently.
    Task<byte[]> downloadTaskA = httpClient.GetByteArrayAsync(urlA);
    Task<byte[]> downloadTaskB = httpClient.GetByteArrayAsync(urlB);

    // Wait for either of the tasks to complete.
    Task<byte[]> completedTask =
        await Task.WhenAny(downloadTaskA, downloadTaskB);

    // Return the length of the data retrieved from that URL.
    byte[] data = await completedTask;
    return data.Length;
}
```

If you are using the `Microsoft.Bcl.Async` NuGet library, the `WhenAny` member is on the `TaskEx` type, not the `Task` type.

Discussion

The task returned by `Task.WhenAny` never completes in a faulted or canceled state. It always results in the first `Task` to complete; if that task completed with an exception, then the exception is not propogated to the task returned by `Task.WhenAny`. For this reason, you should usually `await` the task after it has completed.

When the first task completes, consider whether to cancel the remaining tasks. If the other tasks are not canceled but are also never awaited, then they are abandoned. Abandoned tasks will run to completion, and their results will be ignored. Any exceptions from those abandoned tasks will also be ignored.

It is possible to use `Task.WhenAny` to implement timeouts (e.g., using `Task.Delay` as one of the tasks), but it's not recommended. It's more natural to express timeouts with cancellation, and cancellation has the added benefit that it can actually *cancel* the operation(s) if they time out.

Another antipattern for `Task.WhenAny` is handling tasks as they complete. At first it seems like a reasonable approach to keep a list of tasks and remove each task from the list as it completes. The problem with this approach is that it executes in O(N^2) time, when an O(N) algorithm exists. The proper O(N) algorithm is discussed in Recipe 2.6.

See Also

Recipe 2.4 covers asynchronously waiting for *all* of a collection of tasks to complete.

Recipe 2.6 covers waiting for a collection of tasks to complete and performing actions as each one completes.

Recipe 9.3 covers using a cancellation token to implement timeouts.

2.6. Processing Tasks as They Complete

Problem

You have a collection of tasks to await, and you want to do some processing on each task after it completes. However, you want to do the processing for each one as soon as it completes, not waiting for any of the other tasks.

As an example, this is some code that kicks off three delay tasks and then awaits each one:

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

// Currently, this method prints "2", "3", and "1".
// We want this method to print "1", "2", and "3".
static async Task ProcessTasksAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    // Await each task in order.
    foreach (var task in tasks)
    {
        var result = await task;
        Trace.WriteLine(result);
    }
}
```

The code currently awaits each task in sequence order, even though the second task in the sequence is the first one to complete. What we want is to do the processing (e.g., `Trace.WriteLine`) as each task completes without waiting for the others.

Solution

There are a few different approaches you can take to solve this problem. The one described first in this recipe is the recommended approach; another is described in the Discussion section.

The easiest solution is to restructure the code by introducing a higher-level `async` method that handles awaiting the task and processing its result. Once the processing is factored out, the code is significantly simplified:

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

static async Task AwaitAndProcessAsync(Task<int> task)
{
    var result = await task;
    Trace.WriteLine(result);
```

```
}

// This method now prints "1", "2", and "3".
static async Task ProcessTasksAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    var processingTasks = (from t in tasks
        select AwaitAndProcessAsync(t)).ToArray();

    // Await all processing to complete
    await Task.WhenAll(processingTasks);
}
```

Alternatively, this can be written as:

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

// This method now prints "1", "2", and "3".
static async Task ProcessTasksAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    var processingTasks = tasks.Select(async t =>
    {
        var result = await t;
        Trace.WriteLine(result);
    }).ToArray();

    // Await all processing to complete
    await Task.WhenAll(processingTasks);
}
```

This refactoring is the cleanest and most portable way to solve this problem. However, it is subtly different than the original code. This solution will do the task processing concurrently, whereas the original code would do the task processing one at a time. Most of the time this is not a problem, but if it is not acceptable for your situation, then consider using locks (Recipe 11.2) or the following alternative solution.

Discussion

If refactoring the code like this is not a palatable solution, then there is an alternative. Stephen Toub and Jon Skeet have both developed an extension method that returns an array of tasks that will complete in order. Stephen Toub's solution is available on the Parallel Programming with .NET blog, and Jon Skeet's solution is available on his coding blog.

This extension method is also available in the open source AsyncEx library, available in the `Nito.AsyncEx` NuGet package.

Using an extension method like `OrderByCompletion` minimizes the changes to the original code:

```
static async Task<int> DelayAndReturnAsync(int val)
{
    await Task.Delay(TimeSpan.FromSeconds(val));
    return val;
}

// This method now prints "1", "2", and "3".
static async Task UseOrderByCompletionAsync()
{
    // Create a sequence of tasks.
    Task<int> taskA = DelayAndReturnAsync(2);
    Task<int> taskB = DelayAndReturnAsync(3);
    Task<int> taskC = DelayAndReturnAsync(1);
    var tasks = new[] { taskA, taskB, taskC };

    // Await each one as they complete.
    foreach (var task in tasks.OrderByCompletion())
    {
        var result = await task;
        Trace.WriteLine(result);
    }
}
```

See Also

Recipe 2.4 covers asynchronously waiting for a sequence of tasks to complete.

2.7. Avoiding Context for Continuations

Problem

When an `async` method resumes after an `await`, by default it will resume executing within the same context. This can cause performance problems if that context was a UI context and a large number of `async` methods are resuming on the UI context.

Solution

To avoid resuming on a context, `await` the result of `ConfigureAwait` and pass `false` for its `continueOnCapturedContext` parameter:

```
async Task ResumeOnContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));

    // This method resumes within the same context.
}

async Task ResumeWithoutContextAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);

    // This method discards its context when it resumes.
}
```

Discussion

Having too many continuations run on the UI thread can cause a performance problem. This type of performance problem is difficult to diagnose, since it is not a single method that is slowing down the system. Rather, the UI performance begins to suffer from "thousands of paper cuts" as the application grows more complex.

The real question is, *how many* continuations on the UI thread are *too many?* There is no hard-and-fast answer, but Lucian Wischik of Microsoft has publicized the guideline used by the WinRT team: a hundred or so per second is OK, but a thousand or so per second is too many.

It's best to avoid this right at the beginning. For every `async` method you write, if it doesn't *need* to resume to its original context, then use `ConfigureAwait`. There's no disadvanage to doing so.

It's also a good idea to be context aware when writing `async` code. Normally, an `async` method should *either* require context (dealing with UI elements or ASP.NET requests/responses), or it should be free from context (doing background operations). If you have an `async` method that has parts requiring context and parts context free, consider split-

ting it up into two (or more) `async` methods. This helps keep your code better organized into layers.

See Also

Chapter 1 covers an introduction to asynchronous programming.

2.8. Handling Exceptions from async Task Methods

Problem

Exception handling is a critical part of any design. It's easy to design for the success case but a design is not correct until it also handles the failure cases. Fortunately, handling exceptions from `async Task` methods is straightforward.

Solution

Exceptions can be caught by a simple `try/catch`, just like you would for synchronous code:

```
static async Task ThrowExceptionAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  throw new InvalidOperationException("Test");
}

static async Task TestAsync()
{
  try
  {
    await ThrowExceptionAsync();
  }
  catch (InvalidOperationException)
  {
  }
}
```

Exceptions raised from `async Task` methods are placed on the returned `Task`. They are only raised when the returned `Task` is awaited:

```
static async Task ThrowExceptionAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(1));
  throw new InvalidOperationException("Test");
}

static async Task TestAsync()
{
  // The exception is thrown by the method and placed on the task.
```

```
    Task task = ThrowExceptionAsync();
    try
    {
      // The exception is reraised here, where the task is awaited.
      await task;
    }
    catch (InvalidOperationException)
    {
      // The exception is correctly caught here.
    }
  }
```

Discussion

When an exception is thrown out of an `async Task` method, that exception is captured and put on the returned `Task`. Since `async void` methods don't have a `Task` to put their exception on, their behavior is different; we'll cover that in another recipe.

When you `await` a faulted `Task`, the first exception on that task is rethrown. If you're familiar with the problems of rethrowing exceptions, you may be wondering about stack traces. Rest assured: when the exception is rethrown, the original stack trace is correctly preserved.

This setup sounds somewhat complicated, but all this complexity works together so that the simple scenario has simple code. In the common case, your code should propagate exceptions from asynchronous methods that it calls; all it has to do is `await` the task returned from that asynchronous method, and the exception will be propagated naturally.

There are some situations (such as `Task.WhenAll`) where a `Task` may have multiple exceptions, and `await` will only rethrow the first one. See Recipe 2.4 for an example of handling all exceptions.

See Also

Recipe 2.4 covers waiting for multiple tasks.

Recipe 2.9 covers techniques for catching exceptions from `async void` methods.

Recipe 6.2 covers unit testing exceptions thrown from `async Task` methods.

2.9. Handling Exceptions from async Void Methods

Problem

You have an `async void` method and need to handle exceptions propagated out of that method.

Solution

There is no good solution. If at all possible, change the method to return `Task` instead of `void`. In some situations, this isn't possible; for example, let's say you need to unit test an `ICommand` implementation (which *must* return `void`). In this case, you can provide a `Task`-returning overload of your `Execute` method as such:

```
sealed class MyAsyncCommand : ICommand
{
    async void ICommand.Execute(object parameter)
    {
        await Execute(parameter);
    }

    public async Task Execute(object parameter)
    {
        ... // Asynchronous command implementation goes here.
    }

    ... // Other members (CanExecute, etc)
}
```

It's best to avoid propagating exceptions out of `async void` methods. If you must use an `async void` method, consider wrapping all of its code in a `try` block and handling the exception directly.

There is another solution for handling exceptions from `async void` methods. When an `async void` method propagates an exception, that exception is raised on the `SynchronizationContext` that was active at the time the `async void` method started executing. If your execution environment provides a `SynchronizationContext`, then it usually has a way to handle these top-level exceptions at a global scope. For example, WPF has `Application.DispatcherUnhandledException`, WinRT has `Application.UnhandledException`, and ASP.NET has `Application_Error`.

It is also possible to handle exceptions from `async void` methods by controlling the `SynchronizationContext`. Writing your own `SynchronizationContext` is not trivial, but you can use the `AsyncContext` type from the free `Nito.AsyncEx` NuGet library. `AsyncContext` is particularly useful for applications that do not have a built-in `SynchronizationContext`, such as Console applications and Win32 services. The next example uses `AsyncContext` in a Console application; in this example, the `async` method does return `Task`, but `AsyncContext` also works for `async void` methods:

```
static class Program
{
  static int Main(string[] args)
  {
    try
    {
```

```
      return AsyncContext.Run(() => MainAsync(args));
    }
    catch (Exception ex)
    {
      Console.Error.WriteLine(ex);
      return -1;
    }
  }

  static async Task<int> MainAsync(string[] args)
  {
    ...
  }
}
```

Discussion

One reason to prefer `async Task` over `async void` is that `Task`-returning methods are easier to test. At the very least, overloading `void`-returning methods with `Task`-returning methods will give you a testable API surface.

If you do need to provide your own `SynchronizationContext` type (such as `AsyncContext`), be sure not to install that `SynchronizationContext` on any threads that don't belong to you. As a general rule, you should not place a `SynchronizationContext` on any thread that already has one (such as UI or ASP.NET request threads); nor should you place a `SynchronizationContext` on thread-pool threads. The main thread of a Console application does belong to you, and so do any threads you manually create yourself.

The `AsyncContext` type is in the `Nito.AsyncEx` NuGet package.

See Also

Recipe 2.8 covers exception handling with `async Task` methods.

Recipe 6.3 covers unit testing `async void` methods.

CHAPTER 3

Parallel Basics

In this chapter, we'll cover patterns for parallel programming. Parallel programming is used to split up CPU-bound pieces of work and divide them among multiple threads. These parallel processing recipes only consider CPU-bound work. If you have naturally asynchronous operations (such as I/O-bound work) that you wish to execute in parallel, then see Chapter 2, and Recipe 2.4 in particular.

The parallel processing abstractions covered in this chapter are part of the Task Parallel Library (TPL). This is built in to the .NET framework but is not available on all platforms (see Table 3-1):

Table 3-1. Platform support for TPL

Platform	Parallel support
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✗
Windows Phone SL 7.1	✗
Silverlight 5	✗

3.1. Parallel Processing of Data

Problem

You have a collection of data and you need to perform the same operation on each element of the data. This operation is CPU-bound and may take some time.

Solution

The `Parallel` type contains a `ForEach` method specifically designed for this. This example takes a collection of matrices and rotates them all:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    Parallel.ForEach(matrices, matrix => matrix.Rotate(degrees));
}
```

There are some situations where you'll want to stop the loop early, such as if you encounter an invalid value. This example inverts each matrix, but if an invalid matrix is encountered, it will abort the loop:

```
void InvertMatrices(IEnumerable<Matrix> matrices)
{
    Parallel.ForEach(matrices, (matrix, state) =>
    {
        if (!matrix.IsInvertible)
            state.Stop();
        else
            matrix.Invert();
    });
}
```

A more common situation is when you want the ability to cancel a parallel loop. This is different than stopping the loop; a loop is *stopped* from inside the loop, and it is *canceled* from outside the loop. For example, a cancel button may cancel a `CancellationTokenSource`, canceling a parallel loop like this one:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
        new ParallelOptions { CancellationToken = token },
        matrix => matrix.Rotate(degrees));
}
```

One thing to keep in mind is that each parallel task may run on a different thread, so any shared state must be protected. The following example inverts each matrix and counts the number of matrices that could not be inverted:

```
// Note: this is not the most efficient implementation.
// This is just an example of using a lock to protect shared state.
int InvertMatrices(IEnumerable<Matrix> matrices)
{
    object mutex = new object();
    int nonInvertibleCount = 0;
    Parallel.ForEach(matrices, matrix =>
    {
        if (matrix.IsInvertible)
        {
```

```
                matrix.Invert();
            }
            else
            {
                lock (mutex)
                {
                    ++nonInvertibleCount;
                }
            }
        });
        return nonInvertibleCount;
    }
```

Discussion

The `Parallel.ForEach` method allows parallel processing over a sequence of values. A similar solution is Parallel LINQ (PLINQ). Parallel LINQ provides much of the same capabilities with a LINQ-like syntax. One difference between `Parallel` and PLINQ is that PLINQ assumes it can use all the cores on the computer, while `Parallel` will dynamically react to changing CPU conditions.

`Parallel.ForEach` is a parallel `foreach` loop. If you need to do a parallel `for` loop, the `Parallel` class also supports a `Parallel.For` method. `Parallel.For` is especially useful if you have multiple arrays of data that all take the same index.

See Also

Recipe 3.2 covers aggregating a series of values in parallel, including sums and averages.

Recipe 3.5 covers the basics of PLINQ.

Chapter 9 covers cancellation.

3.2. Parallel Aggregation

Problem

At the conclusion of a parallel operation, you have to aggregate the results. Examples of aggregation are sums, averages, etc.

Solution

The `Parallel` class supports aggregation through the concept of *local values*, which are variables that exist locally within a parallel loop. This means that the body of the loop can just access the value directly, without having to worry about synchronization. When the loop is ready to aggregate each of its local results, it does so with the `localFinal`

ly delegate. Note that the `localFinally` delegate *does* need to synchronize access to the variable that holds the final result. Here's an example of a parallel sum:

```
// Note: this is not the most efficient implementation.
// This is just an example of using a lock to protect shared state.
static int ParallelSum(IEnumerable<int> values)
{
    object mutex = new object();
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue =>
        {
            lock (mutex)
                result += localValue;
        });
    return result;
}
```

Parallel LINQ has more natural aggregation support than the `Parallel` class:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

OK, that was a cheap shot, since PLINQ has built-in support for many common operators (such as `Sum`). PLINQ also has generic aggregation support via the `Aggregate` operator:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Aggregate(
        seed: 0,
        func: (sum, item) => sum + item
    );
}
```

Discussion

If you are already using the `Parallel` class, you may want to use its aggregation support. Otherwise, in most scenarios, the PLINQ support is more expressive and has shorter code.

See Also

Recipe 3.5 covers the basics of PLINQ.

3.3. Parallel Invocation

Problem

You have a number of methods to call in parallel, and these methods are (mostly) independent of each other.

Solution

The `Parallel` class contains a simple `Invoke` member that is designed for this scenario. Here's an example that splits an array in half and processes each half independently:

```
static void ProcessArray(double[] array)
{
    Parallel.Invoke(
        () => ProcessPartialArray(array, 0, array.Length / 2),
        () => ProcessPartialArray(array, array.Length / 2, array.Length)
    );
}

static void ProcessPartialArray(double[] array, int begin, int end)
{
    // CPU-intensive processing...
}
```

You can also pass an array of delegates to the `Parallel.Invoke` method if the number of invocations is not known until runtime:

```
static void DoAction20Times(Action action)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(actions);
}
```

`Parallel.Invoke` supports cancellation just like the other members of the `Parallel` class:

```
static void DoAction20Times(Action action, CancellationToken token)
{
    Action[] actions = Enumerable.Repeat(action, 20).ToArray();
    Parallel.Invoke(new ParallelOptions { CancellationToken = token }, actions);
}
```

Discussion

`Parallel.Invoke` is a great solution for simple parallel invocation. However, it's not a great fit if you want to invoke an action for each item of input data (use `Parallel.ForEach` instead), or if each action produces some output (use Parallel LINQ instead).

See Also

Recipe 3.1 covers `Parallel.ForEach`, which invokes an action for each item of data.

Recipe 3.5 covers Parallel LINQ.

3.4. Dynamic Parallelism

Problem

You have a more complex parallel situation where the structure and number of parallel tasks depends on information known only at runtime.

Solution

The Task Parallel Library (TPL) is centered around the `Task` type. The `Parallel` class and Parallel LINQ are just convenience wrappers around the powerful `Task`. When you need dynamic parallelism, it's easiest to use the `Task` type directly.

Here is one example where some expensive processing needs to be done for each node of a binary tree. The structure of the tree won't be known until runtime, so this is a good scenario for dynamic parallelism. The `Traverse` method processes the current node and then creates two child tasks, one for each branch underneath the node (for this example, we're assuming that the parent nodes must be processed before the children). The `ProcessTree` method starts the processing by creating a top-level parent task and waiting for it to complete:

```
void Traverse(Node current)
{
    DoExpensiveActionOnNode(current);
    if (current.Left != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Left),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
    if (current.Right != null)
    {
        Task.Factory.StartNew(() => Traverse(current.Right),
            CancellationToken.None,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default);
    }
}

public void ProcessTree(Node root)
{
```

```
        var task = Task.Factory.StartNew(() => Traverse(root),
            CancellationToken.None,
            TaskCreationOptions.None,
            TaskScheduler.Default);
        task.Wait();
    }
```

If you don't have a parent/child kind of situation, you can schedule any task to run after another by using a task *continuation*. The continuation is a separate task that executes when the original task completes:

```
Task task = Task.Factory.StartNew(
    () => Thread.Sleep(TimeSpan.FromSeconds(2)),
    CancellationToken.None,
    TaskCreationOptions.None,
    TaskScheduler.Default);
Task continuation = task.ContinueWith(
    t => Trace.WriteLine("Task is done"),
    CancellationToken.None,
    TaskContinuationOptions.None,
    TaskScheduler.Default);
// The "t" argument to the continuation is the same as "task".
```

Discussion

The preceding example code uses `CancellationToken.None` and `TaskScheduler.Default`. Cancellation tokens are covered in Recipe 9.2, and task schedulers are covered in Recipe 12.3. It is always a good idea to explicitly specify the `TaskScheduler` used by `StartNew` and `ContinueWith`.

This arrangement of parent and child tasks is common with dynamic parallelism; however, it is not required. It is equally possible to store each new task in a threadsafe collection and then wait for them all to complete using `Task.WaitAll`.

Using `Task` for parallel processing is completely different than using `Task` for asynchronous processing. See below.

The `Task` type serves two purposes in concurrent programming: it can be a parallel task or an asynchronous task. Parallel tasks may use blocking members, such as `Task.Wait`, `Task.Result`, `Task.WaitAll`, and `Task.WaitAny`. Parallel tasks also commonly use `AttachedToParent` to create parent/child relationships between tasks. Parallel tasks should be created with `Task.Run` or `Task.Factory.StartNew`.

In contrast, asynchronous tasks should avoid blocking members and prefer `await`, `Task.WhenAll`, and `Task.WhenAny`. Asynchronous tasks do not use

`AttachedToParent`, but they can form an implicit kind of parent/child relationship by awaiting another task.

See Also

Recipe 3.3 covers invoking a sequence of methods in parallel, when all the methods are known at the start of the parallel work.

3.5. Parallel LINQ

Problem

You have parallel processing to perform on a sequence of data, producing another sequence of data or a summary of that data.

Solution

Most developers are familiar with LINQ, which you can use to write pull-based calculations over sequences. Parallel LINQ (PLINQ) extends this LINQ support with parallel processing.

PLINQ works well in streaming scenarios, when you have a sequence of inputs and are producing a sequence of outputs. Here's a simple example that just multiplies each element in a sequence by two (real-world scenarios will be much more CPU-intensive than a simple multiply):

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().Select(item => item * 2);
}
```

The example may produce its outputs in any order; this is the default for Parallel LINQ. You can also specify the order to be preserved. The following example is still processed in parallel, but preserves the original order:

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel().AsOrdered().Select(item => item * 2);
}
```

Another natural use of Parallel LINQ is to aggregate or summarize the data in parallel. The following code performs a parallel summation:

```
static int ParallelSum(IEnumerable<int> values)
{
    return values.AsParallel().Sum();
}
```

Discussion

The `Parallel` class is good for many scenarios, but PLINQ code is simpler when doing aggregation or transforming one sequence to another. Bear in mind that the `Parallel` class is more friendly to other processes on the system than PLINQ; this is especially a consideration if the parallel processing is done on a server machine.

PLINQ provides parallel versions of a wide variety of operators, including filtering (`Where`), projection (`Select`), and a variety of aggregations, such as `Sum`, `Average`, and the more generic `Aggregate`. In general, anything you can do with regular LINQ you can do in parallel with PLINQ. This makes PLINQ a great choice if you have existing LINQ code that would benefit from running in parallel.

See Also

Recipe 3.1 covers how to use the `Parallel` class to execute code for each element in a sequence.

Recipe 9.5 covers how to cancel PLINQ queries.

CHAPTER 4
Dataflow Basics

TPL Dataflow is a powerful library that allows you to create a mesh or pipeline and then (asynchronously) send your data through it. Dataflow is a very declarative style of coding; normally, you completely define the mesh first and then start processing data. The mesh ends up being a structure through which your data flows. This requires you to think about your application a bit differently, but once you make that leap, Dataflow becomes a natural fit for many scenarios.

Each mesh is comprised of various blocks that are linked to each other. The individual blocks are simple and are responsible for a single step in the data processing. When a block finishes working on its data, it will pass it along to any linked blocks.

To use TPL Dataflow, install the NuGet package `Microsoft.Tpl.Dataflow` into your application. The TPL Dataflow library has limited platform support for older platforms (Table 4-1):

Table 4-1. Platform support for TPL Dataflow

Platform	Dataflow support
.NET 4.5	✓
.NET 4.0	✗
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	✗
Silverlight 5	✗

4.1. Linking Blocks

Problem

You need to link dataflow blocks into each other to create a mesh.

Solution

The blocks provided by the TPL Dataflow library define only the most basic members. Many of the useful TPL Dataflow methods are actually extension methods. In this case, we're interested in `LinkTo`:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

// After linking, values that exit multiplyBlock will enter subtractBlock.
multiplyBlock.LinkTo(subtractBlock);
```

By default, linked dataflow blocks only propagate data; they do not propagate completion (or errors). If your dataflow is linear (like a pipeline), then you'll probably want to propagate completion. To propagate completion (and errors), you can set the `PropagateCompletion` option on the link:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

var options = new DataflowLinkOptions { PropagateCompletion = true };
multiplyBlock.LinkTo(subtractBlock, options);

...

// The first block's completion is automatically propagated to the second block.
multiplyBlock.Complete();
await subtractBlock.Completion;
```

Discussion

Once linked, data will flow automatically from the source block to the target block. The `PropagateCompletion` option flows completion in addition to data; however, at each step in the pipeline, a faulting block will propagate its exception to the next block wrapped in an `AggregateException`. So, if you have a long pipeline that propagates completions, the original error may be nested within multiple `AggregateException` instances. `AggregateException` has several members, such as `Flatten`, that assist with error handling in this situation.

It is possible to link dataflow blocks in many ways; you can have forks and joins, and even loops in your mesh. However, the simple, linear pipeline is sufficient for most

scenarios. This book will mainly deal with pipelines (and briefly cover forks); more advanced scenarios are beyond the scope of this book.

The `DataflowLinkOptions` type gives you several different options you can set on a link (such as the `PropagateCompletion` option we used above), and the `LinkTo` overload can also take a predicate that you can use to filter which data can go over a link. If data does not pass the filter, it is not dropped. Data that passes the filter travels over that link; data that does not pass the filter attempts to pass over an alternate link, and stays in the block if there is no other link for it to take.

See Also

Recipe 4.2 covers propagating errors along links.

Recipe 4.3 covers removing links between blocks.

Recipe 7.7 covers how to link dataflow blocks to Rx observable streams.

4.2. Propagating Errors

Problem

You need a way to respond to errors that can happen in your dataflow mesh.

Solution

If a delegate passed to a dataflow block throws an exception, then that block will enter a faulted state. When a block is in a faulted state, it will drop all of its data (and stop accepting new data). The block in this code will never produce any output data; the first value raises an exception, and the second value is just dropped:

```
var block = new TransformBlock<int, int>(item =>
{
    if (item == 1)
        throw new InvalidOperationException("Blech.");
    return item * 2;
});
block.Post(1);
block.Post(2);
```

To catch exceptions from a dataflow block, `await` its `Completion` property. The `Completion` property returns a `Task` that will complete when the block is completed, and if the block faults, the `Completion` task is also faulted:

```
try
{
    var block = new TransformBlock<int, int>(item =>
    {
```

```
            if (item == 1)
                throw new InvalidOperationException("Blech.");
            return item * 2;
        });
        block.Post(1);
        await block.Completion;
    }
    catch (InvalidOperationException)
    {
        // The exception is caught here.
    }
```

When you propagate completion using the `PropagateCompletion` link option, errors are also propagated. However, the exception is passed to the next block wrapped in an `AggregateException`. This example catches the exception from the end of a pipeline, so it would catch `AggregateException` if an exception was propagated from earlier blocks:

```
    try
    {
        var multiplyBlock = new TransformBlock<int, int>(item =>
        {
            if (item == 1)
                throw new InvalidOperationException("Blech.");
            return item * 2;
        });
        var subtractBlock = new TransformBlock<int, int>(item => item - 2);
        multiplyBlock.LinkTo(subtractBlock,
            new DataflowLinkOptions { PropagateCompletion = true });
        multiplyBlock.Post(1);
        await subtractBlock.Completion;
    }
    catch (AggregateException)
    {
        // The exception is caught here.
    }
```

Each block wraps incoming errors in an `AggregateException`, even if the incoming error is already an `AggregateException`. If an error occurs early in a pipeline and travels down several links before it is observed, the original error will be wrapped in multiple layers of `AggregateException`. The `AggregateException.Flatten` method simplifies error handling in this scenario.

Discussion

When you build your mesh (or pipeline), consider how errors should be handled. In simpler situations, it can be best to just propagate the errors and catch them once at the end. In more complex meshes, you may need to observe each block when the dataflow has completed.

See Also

Recipe 4.1 covers establishing links between blocks.

Recipe 4.3 covers breaking links between blocks.

4.3. Unlinking Blocks

Problem

During processing, you need to dynamically change the structure of your dataflow. This is an advanced scenario and is hardly ever needed.

Solution

You can link or unlink dataflow blocks at any time; data can be freely passing through the mesh and it is still safe to link or unlink at any time. Both linking and unlinking are fully threadsafe.

When you create a dataflow block link, keep the `IDisposable` returned by the `LinkTo` method, and dispose of it when you want to unlink the blocks:

```
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);

IDisposable link = multiplyBlock.LinkTo(subtractBlock);
multiplyBlock.Post(1);
multiplyBlock.Post(2);

// Unlink the blocks.
// The data posted above may or may not have already gone through the link.
// In real-world code, consider a using block rather than calling Dispose.
link.Dispose();
```

Discussion

Unless you can guarantee that the link is idle, there will be race conditions when you unlink it. However, these race conditions are usually not a concern; data will either flow over the link before the link is broken, or it will not. There are no race conditions that would cause duplication or loss of data.

Unlinking is an advanced scenario, but it can be useful in a handful of situations. As one example, there is no way to change the filter for a link. To change the filter on an existing link, you would have to unlink the old one and create a new link with the new filter (optionally setting `DataflowLinkOptions.Append` to false). As another example, unlinking at a strategic point can be used to pause a dataflow mesh.

See Also

Recipe 4.1 covers establishing links between blocks.

4.4. Throttling Blocks

Problem

You have a fork scenario in your dataflow mesh and want the data to flow in a load-balancing way.

Solution

By default, when a block produces output data, it will examine all of its links (in the order they were created) and attempt to flow the data down each link one at a time. Also, by default, each block will maintain an input buffer and accept any amount of data before it is ready to process it.

This causes a problem in a fork scenario where one source block is linked to two target blocks: the first target block would always buffer the data, and the second target block would never get a chance to get any. This can be fixed by throttling the target blocks using the `BoundedCapacity` block option. By default, `BoundedCapacity` is set to `DataflowBlockOptions.Unbounded`, which causes the first target block to buffer *all* the data even if it is not ready to process it yet.

`BoundedCapacity` can be set to any value greater than zero (or `DataflowBlockOptions.Unbounded`, of course). As long as the target blocks can keep up with the data coming from the source blocks, a simple value of 1 will suffice:

```
var sourceBlock = new BufferBlock<int>();
var options = new DataflowBlockOptions { BoundedCapacity = 1 };
var targetBlockA = new BufferBlock<int>(options);
var targetBlockB = new BufferBlock<int>(options);

sourceBlock.LinkTo(targetBlockA);
sourceBlock.LinkTo(targetBlockB);
```

Discussion

Throttling is useful for load balancing in fork scenarios, but it can be used anywhere else you want throttling behavior. For example, if you are populating your dataflow mesh with data from an I/O operation, you can apply `BoundedCapacity` to the blocks in your mesh. This way, you won't read too much I/O data until your mesh is ready for it, and your mesh won't end up buffering all the input data before it is able to process it.

See Also

Recipe 4.1 covers linking blocks together.

4.5. Parallel Processing with Dataflow Blocks

Problem

You want some parallel processing done within your dataflow mesh.

Solution

By default, each dataflow block is independent from each other block. When you link two blocks together, they will process independently. So, every dataflow mesh has some natural parallelism built in.

If you need to go beyond this—say, if you have one particular block that does heavy CPU computations—then you can instruct that block to operate in parallel on its input data by setting the `MaxDegreeOfParallelism` option. By default, `MaxDegreeOfParallelism` is set to 1, so each dataflow block will only process one piece of data at a time.

`BoundedCapacity` can be set to `DataflowBlockOptions.Unbounded` or any value greater than zero. The following example permits any number of tasks to be multiplying data simultaneously:

```
var multiplyBlock = new TransformBlock<int, int>(
    item => item * 2,
    new ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = DataflowBlockOptions.Unbounded
    }
);
var subtractBlock = new TransformBlock<int, int>(item => item - 2);
multiplyBlock.LinkTo(subtractBlock);
```

Discussion

The `MaxDegreeOfParallelism` option makes parallel processing within a block easy to do. What is not so easy is determining which blocks need it. One technique is to pause dataflow execution in the debugger, where you can see the number of data items queued up (that have not yet been processed by the block). This can be an indication that some restructuring or parallelization would be helpful.

`MaxDegreeOfParallelism` also works if the dataflow block does asynchronous processing. In this case, the `MaxDegreeOfParallelism` option specifies the level of concurrency—a certain number of *slots*. Each data item takes up a slot when the block begins

processing it, and only leaves that slot when the asynchronous processing is fully completed.

See Also

Recipe 4.1 covers linking blocks together.

4.6. Creating Custom Blocks

Problem

You have some reusable logic that you wish to place into a custom dataflow block. This enables you to create larger blocks that contain complex logic.

Solution

You can cut out any part of a dataflow mesh that has a single input and output block by using the `Encapsulate` method. `Encapsulate` will create a single block out of the two endpoints. Propagating data *and completion* between those endpoints is your responsibility. The following code creates a custom dataflow block out of two blocks, propagating data and completion:

```
IPropagatorBlock<int, int> CreateMyCustomBlock()
{
    var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
    var addBlock = new TransformBlock<int, int>(item => item + 2);
    var divideBlock = new TransformBlock<int, int>(item => item / 2);

    var flowCompletion = new DataflowLinkOptions { PropagateCompletion = true };
    multiplyBlock.LinkTo(addBlock, flowCompletion);
    addBlock.LinkTo(divideBlock, flowCompletion);

    return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```

Discussion

When you encapsulate a mesh into a custom block, consider what kind of options you want to expose to your users. Consider how each block option should (or should not) be passed on to your inner mesh; in many cases, some block options don't apply or don't make sense. For this reason, it's common for custom blocks to define their own custom options instead of accepting a `DataflowBlockOptions` parameter.

`DataflowBlock.Encapsulate` will only encapsulate a mesh with one input block and one output block. If you have a reusable mesh with multiple inputs and/or outputs, you should encapsulate it within a custom object and expose the inputs and outputs as

properties of type `ITargetBlock<T>` (for inputs) and `IReceivableSourceBlock<T>` (for outputs).

The previous examples all use `Encapsulate` to create a custom block. It is also possible to implement the dataflow interfaces yourself, but it's much more difficult. Microsoft has a paper that describes advanced techniques for creating your own custom dataflow blocks.

See Also

Recipe 4.1 covers linking blocks together.

Recipe 4.2 covers propagating errors along block links.

CHAPTER 5
Rx Basics

LINQ is a set of language features that enable developers to query sequences. The two most common LINQ providers are the built-in LINQ to Objects (based on `IEnumerable<T>`) and LINQ to Entities (based on `IQueryable<T>`). There are many other providers available, and most providers have the same general structure. Queries are lazily evaluated, and the sequences produce values as necessary. Conceptually, this is a pull model; during evaluation, value items are pulled from the query one at a time.

Reactive Extensions (Rx) treats events as sequences of data that arrive over time. As such, you can think of Rx as LINQ to events (based on `IObservable<T>`). The main difference between observables and other LINQ providers is that Rx is a "push" model. This means that the query defines how the program reacts as events arrive. Rx builds on top of LINQ, adding some powerful new operators as extension methods.

In this chapter, we'll look at some of the more common Rx operations. Bear in mind that all of the LINQ operators are also available, so simple operations, such as filtering (`Where`) and projection (`Select`), work conceptually the same as they do with any other LINQ provider. This chapter will not cover these common LINQ operations; it focuses on the new capabilities that Rx builds on top of LINQ, particularly those dealing with *time*.

To use Rx, install the NuGet package `Rx-Main` into your application. Reactive Extensions has wide platform support (Table 5-1):

Table 5-1. Platform support for Reactive Extensions

Platform	Rx support
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓

Platform	Rx support
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	✓
Silverlight 5	✓

5.1. Converting .NET Events

Problem

You have an event that you need to treat as an Rx input stream, producing some data via `OnNext` each time the event is raised.

Solution

The `Observable` class defines several event converters. Most .NET framework events are compatible with `FromEventPattern`, but if you have events that don't follow the common pattern, you can use `FromEvent` instead.

`FromEventPattern` works best if the event delegate type is `EventHandler<T>`. Many newer framework types use this delegate type for events. For example, the `Progress<T>` type defines a `ProgressChanged` event, which is of type `EventHandler<T>`, so it can be easily wrapped with `FromEventPattern`:

```
var progress = new Progress<int>();
var progressReports = Observable.FromEventPattern<int>(
    handler => progress.ProgressChanged += handler,
    handler => progress.ProgressChanged -= handler);
progressReports.Subscribe(data => Trace.WriteLine("OnNext: " + data.EventArgs));
```

Note that the `data.EventArgs` is strongly typed to be an `int`. The type argument to `FromEventPattern` (`int` in the previous example) is the same as the type `T` in `EventHandler<T>`. The two lambda arguments to `FromEventPattern` enable Rx to subscribe and unsubscribe from the event.

Newer UI frameworks use `EventHandler<T>` and can easily be used with `FromEventPattern`, but older types often define a unique delegate type for each event. These can also be used with `FromEventPattern`, but it takes a bit more work. For example, the `System.Timers.Timer` type defines an `Elapsed` event, which is of type `ElapsedEventHandler`. You can wrap older events like this with `FromEventPattern` as such:

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
var ticks = Observable.FromEventPattern<ElapsedEventHandler, ElapsedEventArgs>(
    handler => (s, a) => handler(s, a),
    handler => timer.Elapsed += handler,
```

```
        handler => timer.Elapsed -= handler);
ticks.Subscribe(data => Trace.WriteLine("OnNext: " + data.EventArgs.SignalTime));
```

Note that `data.EventArgs` is still strongly typed. The type arguments to `FromEventPattern` are now the unique handler type and the derived `EventArgs` type. The first lambda argument to `FromEventPattern` is a converter from `EventHandler<ElapsedEventArgs>` to `ElapsedEventHandler`; the converter should do nothing more than pass along the event.

That syntax is definitely getting awkward. There is another option, which uses reflection:

```
var timer = new System.Timers.Timer(interval: 1000) { Enabled = true };
var ticks = Observable.FromEventPattern(timer, "Elapsed");
ticks.Subscribe(data => Trace.WriteLine("OnNext: "
    + ((ElapsedEventArgs)data.EventArgs).SignalTime));
```

With this approach, the call to `FromEventPattern` is much easier. However, there are some drawbacks to this approach: there is a magic string (`"Elapsed"`), and the consumer does not get strongly typed data. That is, `data.EventArgs` is of type `object`, so you have to cast it to `ElapsedEventArgs` yourself.

Discussion

Events are a common source of data for Rx streams. This recipe covers wrapping any events that conform to the standard event pattern (where the first argument is the sender and the second argument is the event arguments type). If you have unusual event types, you can still use the `Observable.FromEvent` method overloads to wrap them into an observable.

When events are wrapped into an observable, `OnNext` is called each time the event is raised. This can cause surprising behavior when you're dealing with `AsyncCompletedEventArgs`, because any exception is passed along as data (`OnNext`), not as an error (`OnError`). Consider this example wrapper for `WebClient.DownloadStringCompleted`:

```
var client = new WebClient();
var downloadedStrings = Observable.FromEventPattern(client,
    "DownloadStringCompleted");
downloadedStrings.Subscribe(
    data =>
    {
        var eventArgs = (DownloadStringCompletedEventArgs)data.EventArgs;
        if (eventArgs.Error != null)
            Trace.WriteLine("OnNext: (Error) " + eventArgs.Error);
        else
            Trace.WriteLine("OnNext: " + eventArgs.Result);
    },
    ex => Trace.WriteLine("OnError: " + ex.ToString()),
    () => Trace.WriteLine("OnCompleted"));
client.DownloadStringAsync(new Uri("http://invalid.example.com/"));
```

When `WebClient.DownloadStringAsync` completes with an error, the event is raised with an exception in `AsyncCompletedEventArgs.Error`. Unfortunately, Rx sees this as a data event, so if you run this you'll see "OnNext: (Error)" printed instead of "OnError:."

Some event subscriptions and unsubscriptions must be done from a particular context. For example, events on many UI controls must be subscribed to from the UI thread. Rx provides an operator will control the context for subscribing and unsubscribing: `SubscribeOn`. This operator that is not necessary in most situations because most of the time a UI-based subscription is done from the UI thread.

See Also

Recipe 5.2 covers how to change the context in which events are raised.

Recipe 5.4 covers how to throttle events so subscribers are not overwhelmed.

5.2. Sending Notifications to a Context

Problem

Rx does its best to be thread agnostic. So, it will raise its notifications (e.g., `OnNext`) in whatever thread happens to be present at the time.

However, you often want these notifications raised in a particular context. For example, UI elements should only be manipulated from the UI thread that owns them, so if you are updating a UI in response to a notification, then you'll need to "move" over to the UI thread.

Solution

Rx provides the `ObserveOn` operator to move notifications to another scheduler.

Consider this example, which uses the `Interval` operator to create `OnNext` notifications once a second:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

On my machine, the output looks like this:

```
UI thread is 9
Interval 0 on thread 10
Interval 1 on thread 10
```

```
Interval 2 on thread 11
Interval 3 on thread 11
Interval 4 on thread 10
Interval 5 on thread 11
Interval 6 on thread 11
```

Since `Interval` is based on a timer (without a specific thread), the notifications are raised on a thread-pool thread, rather than the UI thread. If we need to update a UI element, we can pipe those notifications through `ObserveOn` and pass a synchronization context representing the UI thread:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.Interval(TimeSpan.FromSeconds(1))
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Interval " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

Another common usage of `ObserveOn` is to move *off* the UI thread when necessary. Let's say we have a situation where we need to do some CPU-intensive computation whenever the mouse moves. By default, all mouse move events are raised on the UI thread, so we can use `ObserveOn` to move those notifications to a thread-pool thread, do the computation, and then move the result notifications back to the UI thread:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var uiContext = SynchronizationContext.Current;
    Trace.WriteLine("UI thread is " + Environment.CurrentManagedThreadId);
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(evt => evt.EventArgs.GetPosition(this))
        .ObserveOn(Scheduler.Default)
        .Select(position =>
        {
            // Complex calculation
            Thread.Sleep(100);
            var result = position.X + position.Y;
            Trace.WriteLine("Calculated result " + result + " on thread " +
                Environment.CurrentManagedThreadId);
            return result;
        })
        .ObserveOn(uiContext)
        .Subscribe(x => Trace.WriteLine("Result " + x + " on thread " +
            Environment.CurrentManagedThreadId));
}
```

If you execute this sample, you'll see the calculations done on a thread-pool thread and the results printed on the UI thread. However, you'll also notice that the calculations and results will lag behind the input; they'll queue up because the mouse move updates more often than every 100 ms. Rx has several techniques for handling this situation; one common one covered in Recipe 5.4 is throttling the input.

Discussion

`ObserveOn` actually moves notifications to an Rx *scheduler*. This recipe covered the default (thread pool) scheduler and one way of creating a UI scheduler. The most common uses for the `ObserveOn` operator are moving on or off the UI thread, but schedulers are useful in other scenarios. We'll take another look at schedulers when we do some advanced testing in Recipe 6.6.

See Also

Recipe 5.1 covers how to create sequences from events.

Recipe 5.4 covers throttling event streams.

Recipe 6.6 covers the special scheduler used for testing your Rx code.

5.3. Grouping Event Data with Windows and Buffers

Problem

You have a sequence of events and you want to group the incoming events as they arrive. For one example, you need to react to pairs of inputs. For another example, you need to react to all inputs within a two-second window.

Solution

Rx provides a pair of operators that group incoming sequences: `Buffer` and `Window`. `Buffer` will hold on to the incoming events until the group is complete, at which time it forwards them all at once as a collection of events. `Window` will logically group the incoming events but will pass them along as they arrive. The return type of `Buffer` is `IObservable<IList<T>>` (an event stream of collections); the return type of `Window` is `IObservable<IObservable<T>>` (an event stream of event streams).

This example uses the `Interval` operator to create `OnNext` notifications once a second and then buffers them two at a time:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Buffer(2)
```

```
            .Subscribe(x => Trace.WriteLine(
                DateTime.Now.Second + ": Got " + x[0] + " and " + x[1]));
    }
```

On my machine, this produces a pair of outputs every two seconds:

```
13: Got 0 and 1
15: Got 2 and 3
17: Got 4 and 5
19: Got 6 and 7
21: Got 8 and 9
```

The following is a similar example using `Window` to create groups of two events:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.Interval(TimeSpan.FromSeconds(1))
        .Window(2)
        .Subscribe(group =>
        {
            Trace.WriteLine(DateTime.Now.Second + ": Starting new group");
            group.Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x),
                () => Trace.WriteLine(DateTime.Now.Second + ": Ending group"));
        });
}
```

On my machine, this `Window` example produces output like this:

```
17: Starting new group
18: Saw 0
19: Saw 1
19: Ending group
19: Starting new group
20: Saw 2
21: Saw 3
21: Ending group
21: Starting new group
22: Saw 4
23: Saw 5
23: Ending group
23: Starting new group
```

These examples illustrate the difference between `Buffer` and `Window`. `Buffer` waits for all the events in its group and then publishes a single collection. `Window` groups events the same way, but publishes the events as they come in.

Both `Buffer` and `Window` also work with time spans. This is an example where all mouse move events are collected in windows of one second:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
```

```
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Buffer(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + x.Count + " items."));
}
```

Depending on how you move the mouse, you should see output like this:

```
49: Saw 93 items.
50: Saw 98 items.
51: Saw 39 items.
52: Saw 0 items.
53: Saw 4 items.
54: Saw 0 items.
55: Saw 58 items.
```

Discussion

`Buffer` and `Window` are some of the tools we have for taming input and shaping it the way we want it to look. Another useful technique is throttling, which we'll look at in Recipe 5.4.

Both `Buffer` and `Window` have other overloads that can be used in more advanced scenarios. The overloads with `skip` and `timeShift` parameters allow you to create groups that overlap other groups or skip elements in between groups. There are also overloads that take delegates, which allow you to dynamically define the boundary of the groups.

See Also

Recipe 5.1 covers how to create sequences from events.

Recipe 5.4 covers throttling event streams.

5.4. Taming Event Streams with Throttling and Sampling

Problem

A common problem with writing reactive code is when the events come in too quickly. A fast-moving stream of events can overwhelm your program's processing.

Solution

Rx provides operators specifically for dealing with a flood of event data. The `Throttle` and `Sample` operators give us two different ways to tame fast input events.

The `Throttle` operator establishes a sliding timeout window. When an incoming event arrives, it resets the timeout window. When the timeout window expires, it publishes the last event value that arrived within the window.

This example monitors mouse movements but uses `Throttle` to only report updates once the mouse has stayed still for a full second:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Throttle(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

The output varies considerably based on mouse movement, but one example run on my machine looked like this:

```
47: Saw 139
49: Saw 137
51: Saw 424
56: Saw 226
```

`Throttle` is often used in situations such as autocomplete, when the user is typing text into a textbox, but you don't want to do the actual lookup until the user stops typing.

`Sample` takes a different approach to taming fast-moving sequences. `Sample` establishes a regular timeout period and publishes the most recent value within that window each time the timeout expires. If there were no values received within the sample period, then no results are published for that period.

The following example captures mouse movements and samples them in one-second intervals. Unlike the `Throttle` example, the `Sample` example does not require you to hold the mouse still to see data.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Sample(TimeSpan.FromSeconds(1))
        .Subscribe(x => Trace.WriteLine(
            DateTime.Now.Second + ": Saw " + (x.X + x.Y)));
}
```

Here's the output on my machine when I first left the mouse still for a few seconds and then continuously moved it:

```
12: Saw 311
17: Saw 254
18: Saw 269
19: Saw 342
20: Saw 224
21: Saw 277
```

Discussion

Throttling and sampling are essential tools for taming the flood of input. Don't forget that you can also easily do filtering with the standard LINQ `Where` operator. You can think of the `Throttle` and `Sample` operators as similar to `Where`, only they filter on time windows instead of filtering on event data. All three of these operators help you tame fast-moving input streams in different ways.

See Also

Recipe 5.1 covers how to create sequences from events.

Recipe 5.2 covers how to change the context in which events are raised.

5.5. Timeouts

Problem

You expect an event to arrive within a certain time and need to ensure that your program will respond in a timely fashion, even if the event does not arrive. Most commonly, this kind of expected event is a single asynchronous operation (e.g., expecting the response from a web service request).

Solution

The `Timeout` operator establishes a sliding timeout window on its input stream. Whenever a new event arrives, the timeout window is reset. If the timeout expires without seeing an event in that window, the `Timeout` operator will end the stream with an `OnError` notification containing a `TimeoutException`.

This example issues a web request for the example domain and applies a timeout of one second:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var client = new HttpClient();
    client.GetStringAsync("http://www.example.com/").ToObservable()
```

```
            .Timeout(TimeSpan.FromSeconds(1))
            .Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + x.Length),
                ex => Trace.WriteLine(ex));
    }
```

`Timeout` is ideal for asynchronous operations, such as web requests, but it can be applied to any event stream. The following example applies `Timeout` to mouse move events, which are easier to play around with:

```
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseMove += handler,
                handler => MouseMove -= handler)
            .Select(x => x.EventArgs.GetPosition(this))
            .Timeout(TimeSpan.FromSeconds(1))
            .Subscribe(
                x => Trace.WriteLine(DateTime.Now.Second + ": Saw " + (x.X + x.Y)),
                ex => Trace.WriteLine(ex));
    }
```

On my machine, I moved the mouse a bit and then let it sit still for a second, and got these results:

```
    16: Saw 180
    16: Saw 178
    16: Saw 177
    16: Saw 176
    System.TimeoutException: The operation has timed out.
```

Note that once the `TimeoutException` is sent to `OnError`, the stream is finished. No more mouse move events come through. You may not want exactly this behavior, so the `Timeout` operator has overloads that substitute a second stream when the timeout occurs instead of ending the stream with an exception.

This example observes mouse moves until there is a timeout and then switches to observing mouse clicks:

```
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        var clicks = Observable.FromEventPattern
            <MouseButtonEventHandler, MouseButtonEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseDown += handler,
                handler => MouseDown -= handler)
            .Select(x => x.EventArgs.GetPosition(this));

        Observable.FromEventPattern<MouseEventHandler, MouseEventArgs>(
                handler => (s, a) => handler(s, a),
                handler => MouseMove += handler,
```

```
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this))
        .Timeout(TimeSpan.FromSeconds(1), clicks)
        .Subscribe(
            x => Trace.WriteLine(
                DateTime.Now.Second + ": Saw " + x.X + "," + x.Y),
            ex => Trace.WriteLine(ex));
}
```

On my machine, I moved the mouse a bit, then held it still for a second, and then clicked on a couple different points. The output is below, showing the mouse-move events quickly moving through until the timout, and then the two click events:

```
49: Saw 95,39
49: Saw 94,39
49: Saw 94,38
49: Saw 94,37
53: Saw 130,141
55: Saw 469,4
```

Discussion

`Timeout` is an essential operator in nontrivial applications because you always want your program to be responsive even if the rest of the world is not. It's particularly useful when you have asynchronous operations, but it can be applied to any event stream. Note that the underlying operation is not actually canceled; in the case of a timeout, the operation will continue executing until it succeeds or fails.

See Also

Recipe 5.1 covers how to create sequences from events.

Recipe 7.6 covers wrapping asynchronous code as an observable event stream.

Recipe 9.6 covers unsubscribing from sequences as a result of a `CancellationToken`.

Recipe 9.3 covers using a `CancellationToken` as a timeout.

CHAPTER 6
Testing

Testing is an essential part of software quality. Unit testing advocates in particular have become common in the last few years; it seems that you read or hear about it everywhere. Some promote test-driven development, which is a style of coding that ensures you have comprehensive tests when the application is complete. The benefits of unit testing on code quality and overall time to completion are well known, and yet (at the time of writing) most developers do not actually write unit tests.

I encourage you to write at least some unit tests, and start with the code where you feel the least confidence. In my personal experience, unit tests have given me two main advantages:

1. Better understanding of the code. You know that part of the application that works but you have no idea how? It's always kind of in the back of your mind when the really weird bug reports come in. Writing unit tests for the "hard" code is a great way to get a clear understanding of how it works. After writing unit tests describing its behavior, the code is no longer mysterious; you end up with a set of unit tests that describe its behavior as well as the dependencies that code has on the rest of the code.
2. Greater confidence to make changes. Sooner or later, you'll get that feature request that requires you to change the "scary" code, and you'll no longer be able to pretend it isn't there (I know how that feels; I've been there!). It's best to be proactive: write the unit tests for the scary code before the feature request comes in; do it now rather than later. Once your unit tests are complete, you'll have an early warning system that will alert you immediately if your changes break existing behavior.

Both of these advantages apply to your own code just as much as others' code. I'm sure there are other advantages, too. Does unit testing decrease the frequency of bugs? Most likely. Does unit testing reduce the overall time on a project? Possibly. But the advantages I describe above are defininte; I experience them every time I write unit tests.

So, that's my sales pitch for unit testing.

This chapter contains recipes that are all about testing. A lot of developers (even ones who normally write unit tests) shy away from testing concurrent code because they assume it's hard. However, as these recipes will show, unit testing concurrent code is not as difficult as they think. Modern features and libraries, such as `async` and Rx, have put a lot of thought into testing, and it shows. I encourage you to use these recipes to write unit tests, especially if you're new to concurrency (i.e., the new concurrent code is "hard" and/or "scary").

6.1. Unit Testing async Methods

Problem

You have an `async` method that you need to unit test.

Solution

Most modern unit test frameworks support `async Task` unit test methods, including MSTest, NUnit, and xUnit. MSTest began support for these tests with Visual Studio 2012. If you use another unit test framework, you may have to upgrade to the latest version.

Here is an example of an `async` MSTest unit test:

```
[TestMethod]
public async Task MyMethodAsync_ReturnsFalse()
{
  var objectUnderTest = ...;
  bool result = await objectUnderTest.MyMethodAsync();
  Assert.IsFalse(result);
}
```

The unit test framework will notice that the return type of the method is `Task` and will intelligently wait for the task to complete before marking the test "successful" or "failed."

If your unit test framework does not support `async Task` unit tests, then it will need some help to wait for the asynchronous operation under test. One option is to use `Task.Wait` and unwrap the `AggregateException` if there is an error. I prefer to use the `AsyncContext` type from the `Nito.AsyncEx` NuGet package:

```
[TestMethod]
public void MyMethodAsync_ReturnsFalse()
{
  AsyncContext.Run(async () =>
  {
    var objectUnderTest = ...;
    bool result = await objectUnderTest.MyMethodAsync();
```

```
      Assert.IsFalse(result);
    });
  }
```

`AsyncContext.Run` will wait until all asynchronous methods complete.

Discussion

Mocking asynchronous dependencies can be a bit awkward at first. It's a good idea to at least test how your methods respond to synchronous success (mocking with `Task.FromResult`), synchronous errors (mocking with `TaskCompletionSource<T>`), and asynchronous success (mocking with `Task.Yield` and a return value).

When testing asynchronous code, deadlocks and race conditions may surface more often than when testing synchronous code. I find the per-test timeout setting useful; in Visual Studio, you can add a test settings file to your solution that allows you to set individual test timeouts. The default value is quite high; I usually have a per-test timeout setting of two seconds.

The `AsyncContext` type is in the `Nito.AsyncEx` NuGet package.

See Also

Recipe 6.2 covers unit testing asynchronous methods expected to fail.

6.2. Unit Testing async Methods Expected to Fail

Problem

You need to write a unit test that checks for a specific failure of an `async Task` method.

Solution

If you're doing desktop or server development, MSTest does support failure testing via the regular `ExpectedExceptionAttribute`:

```
// Not a recommended solution; see below.
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
  await MyClass.DivideAsync(4, 0);
}
```

However, this solution is not the best solution. For one thing, Windows Store applications do not have `ExpectedException` available for their unit tests. Another more philosophical problem is that `ExpectedException` is actually a poor design. The exception it expects may be thrown by *any* of the methods called by your unit test method. A better design checks that a *particular* piece of code throws that exception, not the unit test as a whole.

Microsoft has moved in this direction by removing `ExpectedException` from Windows Store unit tests, replacing it with `Assert.ThrowsException<TException>`. You can use it like this:

```
[TestMethod]
public async Task Divide_WhenDenominatorIsZero_ThrowsDivideByZero()
{
  await Assert.ThrowsException<DivideByZeroException>(async () =>
  {
    await MyClass.DivideAsync(4, 0);
  });
}
```

Do not forget to `await` the task returned by `ThrowsException`! This will propagate any assertion failures that it detects. If you forget the `await` and ignore the compiler warning, your unit test will always silently succeed regardless of your method's behavior.

Unfortunately, Microsoft only added `ThrowsException` to Windows Store unit test projects, and (as of this writing) several other unit test frameworks do not include an equivalent `async`-compatible `ThrowsException`. If you find yourself in this boat, you can create your own:

```
/// <summary>
/// Ensures that an asynchronous delegate throws an exception.
/// </summary>
/// <typeparam name="TException">
/// The type of exception to expect.
/// </typeparam>
/// <param name="action">The asynchronous delegate to test.</param>
/// <param name="allowDerivedTypes">
/// Whether derived types should be accepted.
/// </param>
public static async Task ThrowsExceptionAsync<TException>(Func<Task> action,
    bool allowDerivedTypes = true)
{
    try
    {
        await action();
        Assert.Fail("Delegate did not throw expected exception " +
            typeof(TException).Name + ".");
```

```
        }
        catch (Exception ex)
        {
            if (allowDerivedTypes && !(ex is TException))
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name +
                    " or a derived type was expected.");
            if (!allowDerivedTypes && ex.GetType() != typeof(TException))
                Assert.Fail("Delegate threw exception of type " + ex.GetType().Name +
                    ", but " + typeof(TException).Name + " was expected.");
        }
    }
```

You can use the method just like the Windows Store MSTest `Assert.ThrowsException<TException>` method. Don't forget to `await` the return value!

Discussion

Testing error handling is just as important as testing the successful scenarios. Some would even say more important, since the successful scenario is the one that everyone tries before the software is released. If your application is going to behave strangely, it will be due to an unexpected error situation.

However, I encourage developers to move away from `ExpectedException`. It's better to test for an exception thrown at a specific point rather than testing for an exception at any time during the test. Instead of `ExpectedException`, use `ThrowsException` (or its equivalent in your unit test framework), or use the `ThrowsExceptionAsync` implementation above.

See Also

Recipe 6.1 covers the basics of unit testing asynchronous methods.

6.3. Unit Testing async void Methods

Problem

You have an `async void` method that you need to unit test.

Solution

Stop.

You should do your dead-level best to avoid this problem rather than solve it. If it is possible to change your `async void` method to an `async Task` method, then do so.

If your method *must* be `async void` (e.g., to satisfy an interface method signature), then consider writing two methods: an `async Task` method that contains all the logic, and an `async void` wrapper that just calls the `async Task` method and awaits the result. The `async void` method satisfies the architecture requirements, while the `async Task` method (with all the logic) is testable.

If it's completely impossible to change your method and you absolutely *must* unit test an `async void` method, then there is a way to do it. You can use the `AsyncContext` class from the `Nito.AsyncEx` library:

```
// Not a recommended solution; see above.
[TestMethod]
public void MyMethodAsync_DoesNotThrow()
{
  AsyncContext.Run(() =>
  {
    var objectUnderTest = ...;
    objectUnderTest.MyMethodAsync();
  });
}
```

The `AsyncContext` type will wait until all asynchronous operations complete (including `async void` methods) and will propagate exceptions that they raise.

The `AsyncContext` type is in the `Nito.AsyncEx` NuGet package.

Discussion

One of the key guidelines in `async` code is to avoid `async void`. I strongly recommend you refactor your code instead of using `AsyncContext` for unit testing `async void` methods.

See Also

Recipe 6.1 covers unit testing `async Task` methods.

6.4. Unit Testing Dataflow Meshes

Problem

You have a dataflow mesh in your application, and you need to verify it works correctly.

Solution

Dataflow meshes are independent: they have a lifetime of their own and are asynchronous by nature. So, the most natural way to test them is with an asynchronous unit test. The following unit test verifies the custom dataflow block from Recipe 4.6:

```
[TestMethod]
public async Task MyCustomBlock_AddsOneToDataItems()
{
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Complete();

    Assert.AreEqual(4, myCustomBlock.Receive());
    Assert.AreEqual(14, myCustomBlock.Receive());
    await myCustomBlock.Completion;
}
```

Unit testing failures is not quite as straightforward, unfortunately. This is because exceptions in dataflow meshes are wrapped in another `AggregateException` each time they are propagated to the next block. The following example uses a helper method to ensure that an exception will discard data and propagate through the custom block:

```
[TestMethod]
public async Task MyCustomBlock_Fault_DiscardsDataAndFaults()
{
    var myCustomBlock = CreateMyCustomBlock();

    myCustomBlock.Post(3);
    myCustomBlock.Post(13);
    myCustomBlock.Fault(new InvalidOperationException());

    try
    {
        await myCustomBlock.Completion;
    }
    catch (AggregateException ex)
    {
        AssertExceptionIs<InvalidOperationException>(
            ex.Flatten().InnerException, false);
    }
}

public static void AssertExceptionIs<TException>(Exception ex,
    bool allowDerivedTypes = true)
{
    if (allowDerivedTypes && !(ex is TException))
        Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
        + typeof(TException).Name + " or a derived type was expected.");
    if (!allowDerivedTypes && ex.GetType() != typeof(TException))
```

```
            Assert.Fail("Exception is of type " + ex.GetType().Name + ", but "
            + typeof(TException).Name + " was expected.");
    }
```

Discussion

Unit testing of dataflow meshes directly is doable, but somewhat awkward. If your mesh is a part of a larger component, then you may find that it's easier to just unit test the larger component (implicitly testing the mesh). But if you're developing a reusable custom block or mesh, then unit tests like the preceding ones should be used.

See Also

Recipe 6.1 covers unit testing `async` methods.

6.5. Unit Testing Rx Observables

Problem

Part of your program is using `IObservable<T>`, and you need to find a way to unit test it.

Solution

Reactive Extensions has a number of operators that produce sequences (e.g., `Return`) and other operators that can convert a reactive sequence into a regular collection or item (e.g., `SingleAsync`). We will use operators like `Return` to create stubs for observable dependencies, and operators like `SingleAsync` to test the output.

Consider the following code, which takes an HTTP service as a dependency and applies a timeout to the HTTP call:

```
public interface IHttpService
{
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url)
    {
```

```
            return _httpService.GetString(url)
                .Timeout(TimeSpan.FromSeconds(1));
        }
    }
```

The code we wish to test is `MyTimeoutClass`, which consumes an observable dependency and produces an observable as output.

The `Return` operator creates a cold sequence with a single element in it; we can use this to build a simple stub. The `SingleAsync` operator returns a `Task<T>` that is completed when the next event arrives. `SingleAsync` can be used for simple unit tests like this:

```
class SuccessHttpServiceStub : IHttpService
{
    public IObservable<string> GetString(string url)
    {
        return Observable.Return("stub");
    }
}

[TestMethod]
public async Task MyTimeoutClass_SuccessfulGet_ReturnsResult()
{
    var stub = new SuccessHttpServiceStub();
    var my = new MyTimeoutClass(stub);

    var result = await my.GetStringWithTimeout("http://www.example.com/")
        .SingleAsync();

    Assert.AreEqual("stub", result);
}
```

Another operator important in stub code is `Throw`, which returns an observable that ends with an error. This allows us to unit test the error case as well. The following example uses the `ThrowsExceptionAsync` helper from Recipe 6.2:

```
private class FailureHttpServiceStub : IHttpService
{
    public IObservable<string> GetString(string url)
    {
        return Observable.Throw<string>(new HttpRequestException());
    }
}

[TestMethod]
public async Task MyTimeoutClass_FailedGet_PropagatesFailure()
{
    var stub = new FailureHttpServiceStub();
    var my = new MyTimeoutClass(stub);

    await ThrowsExceptionAsync<HttpRequestException>(async () =>
    {
```

```
            await my.GetStringWithTimeout("http://www.example.com/")
                .SingleAsync();
        });
    }
```

Discussion

`Return` and `Throw` are great for creating observable stubs, and `SingleAsync` is an easy way to test observables with `async` unit tests. They're a good combination for simple observables, but they don't hold up well once you start dealing with *time*. For example, if we wanted to test the timeout capability of `MyTimeoutClass`, the unit test would actually have to wait for that amount of time. This doesn't scale well as we add more unit tests. In Recipe 6.6, we'll look at a special way that Reactive Extensions allow us to stub out time itself.

See Also

Recipe 6.1 covers unit testing `async` methods, which is very similar to unit tests that await `SingleAsync`.

Recipe 6.6 covers unit testing observable sequences that depend on time passing.

6.6. Unit Testing Rx Observables with Faked Scheduling

Problem

You have an observable that is dependent on time, and want to write a unit test that is not dependent on time. Observables that depend on time include ones that use timeouts, windowing/buffering, and throttling/sampling. You want to unit test these but do not want your unit tests to have excessive runtimes.

Solution

It's certainly possible to put delays in your unit tests; however, there are two problems with that approach: 1) the unit tests take a long time to run, and 2) there are race conditions because the unit tests all run at the same time, making timing unpredictable.

The Rx library was designed with testing in mind; in fact, the Rx library itself is extensively unit tested. To enable this, Rx introduced a concept called a *scheduler*, and *every* Rx operator that deals with time is implemented using this abstract scheduler.

To make your observables testable, you need to allow your caller to specify the scheduler. For example, we can take the `MyTimeoutClass` from Recipe 6.5 and add a scheduler:

```
public interface IHttpService
{
```

```
    IObservable<string> GetString(string url);
}

public class MyTimeoutClass
{
    private readonly IHttpService _httpService;

    public MyTimeoutClass(IHttpService httpService)
    {
        _httpService = httpService;
    }

    public IObservable<string> GetStringWithTimeout(string url,
        IScheduler scheduler = null)
    {
        return _httpService.GetString(url)
            .Timeout(TimeSpan.FromSeconds(1), scheduler ?? Scheduler.Default);
    }
}
```

Next, let's modify our HTTP service stub so that it also understands scheduling, and we'll introduce a variable delay:

```
private class SuccessHttpServiceStub : IHttpService
{
    public IScheduler Scheduler { get; set; }
    public TimeSpan Delay { get; set; }

    public IObservable<string> GetString(string url)
    {
        return Observable.Return("stub")
            .Delay(Delay, Scheduler);
    }
}
```

Now you can use `TestScheduler`, a type included in the Rx library. `TestScheduler` gives you powerful control over (virtual) time.

`TestScheduler` is in a separate NuGet package from the rest of Rx; you'll need to install the `Rx-Testing` NuGet package.

`TestScheduler` gives you complete control over time, but you often just need to set up your code and then call `TestScheduler.Start`. `Start` will virtually advance time until everything is done. A simple success test case could look like this:

```
[TestMethod]
public void MyTimeoutClass_SuccessfulGetShortDelay_ReturnsResult()
{
```

```
        var scheduler = new TestScheduler();
        var stub = new SuccessHttpServiceStub
        {
            Scheduler = scheduler,
            Delay = TimeSpan.FromSeconds(0.5),
        };
        var my = new MyTimeoutClass(stub);
        string result = null;

        my.GetStringWithTimeout("http://www.example.com/", scheduler)
            .Subscribe(r => { result = r; });

        scheduler.Start();

        Assert.AreEqual("stub", result);
    }
```

The code simulates a network delay of half a second. It's important to note that this unit test *does not* take half a second to run; on my machine, it takes about 70 milliseconds. The half-second delay only exists in virtual time. The other notable difference in this unit test is that it is not asynchronous; since we are using `TestScheduler`, all our tests can complete immediately.

Now that everything is using test schedulers, it's easy to test timeout situations:

```
    [TestMethod]
    public void MyTimeoutClass_SuccessfulGetLongDelay_ThrowsTimeoutException()
    {
        var scheduler = new TestScheduler();
        var stub = new SuccessHttpServiceStub
        {
            Scheduler = scheduler,
            Delay = TimeSpan.FromSeconds(1.5),
        };
        var my = new MyTimeoutClass(stub);
        Exception result = null;

        my.GetStringWithTimeout("http://www.example.com/", scheduler)
            .Subscribe(_ => Assert.Fail("Received value"), ex => { result = ex; });

        scheduler.Start();

        Assert.IsInstanceOfType(result, typeof(TimeoutException));
    }
```

Once again, this unit test does not take 1 second (or 1.5 seconds) to run; it executes immediately using virtual time.

Discussion

We've just scratched the surface on Reactive Extensions schedulers and virtual time. I recommend that you start unit testing when you start writing Rx code; as your code grows more complex, rest assured that the Rx testing is capable of handling it.

`TestScheduler` also has `AdvanceTo` and `AdvanceBy` methods, which allow you to gradually step through virtual time. There may be situations where these are useful, but you should strive to have your unit tests only test one thing. For example, when testing a timeout, you could write a single unit test that advanced the `TestScheduler` partially and ensured the timeout did not happen early and then advanced the `TestScheduler` past the timeout value and ensured the timeout did happen. However, I prefer to have independent unit tests as much as possible, for example, one unit test ensuring that the timeout did not happen early, and a different unit test ensuring that the timeout did happen later.

See Also

Recipe 6.5 covers the basics of unit testing observable sequences.

CHAPTER 7
Interop

Asynchronous, parallel, reactive—each has its place, but how well do they work together?

In this chapter, we'll look at various *interop* scenarios where we will learn how to combine these different approaches. We'll learn that they complement each other, rather than compete; there is very little friction at the boundaries where one approach meets another.

7.1. Async Wrappers for "Async" Methods with "Completed" Events

Problem

There is an older asynchronous pattern that uses methods named *Operation*`Async` along with events named *Operation*`Completed`. You wish to perform an operation like this and `await` the result.

The *Operation*`Async` and *Operation*`Completed` pattern is called the Event-based Asynchronous Pattern (EAP). We're going to wrap those into a `Task`-returning method that follows the Task-based Asynchronous Pattern (TAP).

Solution

You can create wrappers for asynchronous operations by using the `TaskCompletionSource<TResult>` type. This type controls a `Task<TResult>` and allows you to complete the task at the appropriate time.

The following example defines an extension method for `WebClient` that downloads a `string`. The `WebClient` type defines `DownloadStringAsync` and `DownloadStringCompleted`. Using those, we can define a `DownloadStringTaskAsync` method as such:

```
public static Task<string> DownloadStringTaskAsync(this WebClient client,
    Uri address)
{
    var tcs = new TaskCompletionSource<string>();

    // The event handler will complete the task and unregister itself.
    DownloadStringCompletedEventHandler handler = null;
    handler = (_, e) =>
    {
        client.DownloadStringCompleted -= handler;
        if (e.Cancelled)
            tcs.TrySetCanceled();
        else if (e.Error != null)
            tcs.TrySetException(e.Error);
        else
            tcs.TrySetResult(e.Result);
    };

    // Register for the event and *then* start the operation.
    client.DownloadStringCompleted += handler;
    client.DownloadStringAsync(address);

    return tcs.Task;
}
```

If you're already using the `Nito.AsyncEx` NuGet library, wrappers like this are slightly simpler due to the `TryCompleteFromEventArgs` extension method in that library:

```
public static Task<string> DownloadStringTaskAsync(this WebClient client,
    Uri address)
{
    var tcs = new TaskCompletionSource<string>();

    // The event handler will complete the task and unregister itself.
    DownloadStringCompletedEventHandler handler = null;
    handler = (_, e) =>
    {
        client.DownloadStringCompleted -= handler;
        tcs.TryCompleteFromEventArgs(e, () => e.Result);
    };

    // Register for the event and *then* start the operation.
    client.DownloadStringCompleted += handler;
    client.DownloadStringAsync(address);

    return tcs.Task;
}
```

Discussion

This particular example isn't very useful because `WebClient` already defines a `DownloadStringTaskAsync` and there is a more `async`-friendly `HttpClient` that could also be used. However, this same technique can be used to interface with older asynchronous code that has not yet been updated to use `Task`.

For new code, always use `HttpClient`. Only use `WebClient` if you're working with legacy code.

Normally, a TAP method for downloading strings would be named *`Operation`*`Async` (e.g., `DownloadStringAsync`); however, that naming convention won't work in this case because EAP already defines a method with that name. In this case, the convention is to name the TAP method *`Operation`*`TaskAsync` (e.g., `DownloadStringTaskAsync`).

When wrapping EAP methods, there is the possibility that the "start" method may throw an exception; in the previous example, `DownloadStringAsync` may throw. In that case, you'll need to decide whether to allow the exception to propagate, or catch the exception and call `TrySetException`. Most of the time, exceptions thrown at that point are usage errors, so it doesn't matter which option you choose.

See Also

Recipe 7.2 covers TAP wrappers for APM methods (`Begin`*`Operation`* and `End`*`Operation`*).

Recipe 7.3 covers TAP wrappers for any kind of notification.

7.2. Async Wrappers for "Begin/End" methods

Problem

There is an older asynchronous pattern that uses pairs of methods named `Begin`*`Operation`* and `End`*`Operation`*, with the `IAsyncResult` representing the asynchronous operation. You have an operation that follows this pattern and wish to consume it with `await`.

The `Begin`*`Operation`* and `End`*`Operation`* pattern is called the Asynchronous Programming Model (APM). We're going to wrap those into a `Task`-returning method that follows the Task-based Asynchronous Pattern (TAP).

Solution

The best approach for wrapping APM is to use one of the `FromAsync` methods on the `TaskFactory` type. `FromAsync` uses `TaskCompletionSource<TResult>` under the hood, but when you're wrapping APM, `FromAsync` is much easier to use.

The following example defines an extension method for `WebRequest` that sends an HTTP request and gets the response. The `WebRequest` type defines `BeginGetResponse` and `EndGetResponse`; we can define a `GetResponseAsync` method as such:

```
public static Task<WebResponse> GetResponseAsync(this WebRequest client)
{
    return Task<WebResponse>.Factory.FromAsync(client.BeginGetResponse,
        client.EndGetResponse, null);
}
```

Discussion

`FromAsync` has a downright confusing number of overloads!

As a general rule, it's best to call `FromAsync` like the example. First, pass the `BeginOperation` method (without calling it) then the `EndOperation` method (without calling it). Next, pass all arguments that `BeginOperation`, takes except for the last `AsyncCallback` and `object` arguments. Finally, pass `null`.

In particular, do not call the `BeginOperation` method before calling `FromAsync`. It is possible to call `FromAsync`, passing the `IAsyncOperation` that you get from `BeginOperation`, but if you call it this way, `FromAsync` is forced to use a less performant implementation.

You might be wondering why the recommended pattern always passes a `null` at the end. `FromAsync` was introduced along with the `Task` type in .NET 4.0, before `async` was around. At that time, it was common to use `state` objects in asynchronous callbacks, and the `Task` type supports this via its `AsyncState` member. In the new `async` pattern, state objects are no longer necessary.

See Also

Recipe 7.3 covers writing TAP wrappers for any kind of notification.

7.3. Async Wrappers for Anything

Problem

You have an unusual or nonstandard asynchronous operation or event and wish to consume it via `await`.

Solution

The `TaskCompletionSource<T>` type can be used to construct `Task<T>` objects in any scenario. Using a `TaskCompletionSource<T>`, you can complete a task in three different ways: with a successful result, faulted, or canceled.

Before `async` was on the scene, there were two other asynchronous patterns recommended by Microsoft: APM (which we looked at in Recipe 7.2) and EAP (Recipe 7.1). However, both APM and EAP were rather awkward and in some cases difficult to get right. So, an unofficial convention arose that used callbacks, with methods like this:

```
public interface IMyAsyncHttpService
{
    void DownloadString(Uri address, Action<string, Exception> callback);
}
```

Methods like these follow the convention that `DownloadString` will start the (asynchronous) download, and when it completes, the `callback` is invoked with either the result or the exception. Usually, `callback` is invoked on a background thread.

This nonstandard kind of asynchronous method can also be wrapped using `TaskCompletionSource<T>` so that it naturally works with `await`:

```
public static Task<string> DownloadStringAsync(
    this IMyAsyncHttpService httpService, Uri address)
{
    var tcs = new TaskCompletionSource<string>();
    httpService.DownloadString(address, (result, exception) =>
    {
        if (exception != null)
            tcs.TrySetException(exception);
        else
            tcs.TrySetResult(result);
    });
    return tcs.Task;
}
```

Discussion

This same pattern can be used with `TaskCompletionSource<T>` to wrap *any* asynchronous method, no matter how nonstandard. Create the `TaskCompletionSource<T>` instance first. Next, arrange a callback so that the `TaskCompletionSource<T>` completes its task appropriately. Then, start the actual asynchronous operation. Finally, return the `Task<T>` that is attached to that `TaskCompletionSource<T>`.

One important aspect of this pattern is that you must make *sure* that the `TaskCompletionSource<T>` is always completed. Think through your error handling in particular, and ensure that the `TaskCompletionSource<T>` will be completed appropriately. In the last example, exceptions are explicitly passed into the callback, so we don't need a `catch`

block; but some nonstandard patterns might need you to catch exceptions in your callbacks and place them on the `TaskCompletionSource<T>`.

See Also

Recipe 7.1 covers TAP wrappers for EAP members (*`Operation`*`Async`, *`Operation`*`Completed`).

Recipe 7.2 covers TAP wrappers for APM members (`Begin`*`Operation`*, `End`*`Operation`*).

7.4. Async Wrappers for Parallel Code

Problem

You have (CPU-bound) parallel processing that you wish to consume using `await`. Usually, this is desirable so that your UI thread does not block waiting for the parallel processing to complete.

Solution

The `Parallel` type and Parallel LINQ use the thread pool to do parallel processing. They will also include the calling thread as one of the parallel processing threads, so if you call a parallel method from the UI thread, the UI will be unresponsive until the processing completes.

To keep the UI responsive, wrap the parallel processing in a `Task.Run` and `await` the result:

```
await Task.Run(() => Parallel.ForEach(...));
```

The key behind this recipe is that parallel code *includes the calling thread* in its pool of threads that it uses to do the parallel processing. This is true for both Parallel LINQ and the `Parallel` class.

Discussion

This is a simple recipe but one that is often overlooked. By using `Task.Run`, you are pushing all of the parallel processing off to the thread pool. `Task.Run` returns a `Task` that then represents that parallel work, and the UI thread can (asynchronously) wait for it to complete.

This recipe only applies to UI code. On the server side (e.g., ASP.NET), parallel processing is rarely done. Even if you do perform parallel processing, you should invoke it directly, not push it off to the thread pool.

See Also

Chapter 3 covers the basics of parallel code.

Chapter 2 covers the basics of asynchronous code.

7.5. Async Wrappers for Rx Observables

Problem

You have an observable stream that you wish to consume using `await`.

Solution

First, you need to decide *which* of the observable events in the event stream you're interested in. The common situations are:

- The last event before the stream ends
- The next event
- All the events

To capture the *last* event in the stream, you can either `await` the result of `LastAsync` or just `await` the observable directly:

```
IObservable<int> observable = ...;
int lastElement = await observable.LastAsync();
// or:  int lastElement = await observable;
```

When you `await` an observable or `LastAsync`, the code (asynchronously) waits until the stream completes and then returns the last element. Under the covers, the `await` is subscribing to the stream.

To capture the *next* event in the stream, use `FirstAsync`. In this case, the `await` subscribes to the stream and then completes (and unsubscribes) as soon as the first event arrives:

```
IObservable<int> observable = ...;
int nextElement = await observable.FirstAsync();
```

To capture *all* events in the stream, you can use `ToList`:

```
IObservable<int> observable = ...;
IList<int> allElements = await observable.ToList();
```

Discussion

The Rx library provides all the tools you need to consume streams using `await`. The only tricky part is that you have to think about whether the awaitable will wait until the

stream completes. Of the examples in this recipe, `LastAsync`, `ToList`, and the direct `await` will wait until the stream completes; `FirstAsync` will only wait for the next event.

If these examples don't satisfy your needs, remember that you have the full power of LINQ as well as the new Rx manipulators. Operators such as `Take` and `Buffer` can also help you asynchronously wait for the elements you need without having to wait for the entire stream to complete.

Some of the operators for use with `await`—such as `FirstAsync` and `LastAsync`—do not actually return a `Task<T>`. If you plan to use `Task.WhenAll` or `Task.WhenAny`, then you'll need an actual `Task<T>`, which you can get by calling `ToTask` on any observable. `ToTask` will return a `Task<T>` that completes with the last value in the stream.

See Also

Recipe 7.6 covers using asynchronous code within an observable stream.

Recipe 7.7 covers using observable streams as an input to a dataflow block (which can perform asynchronous work).

Recipe 5.3 covers windows and buffering for observable streams.

7.6. Rx Observable Wrappers for async Code

Problem

You have an asynchronous operation that you want to combine with an observable.

Solution

Any asynchronous operation can be treated as an observable stream that either:

- Produces a single element and then completes
- Faults without producing any elements

The Rx library includes a simple conversion from `Task<T>` to `IObservable<T>` that implements this transformation. The following code starts an asynchronous download of a web page, treating it as an observable sequence:

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response =
    client.GetAsync("http://www.example.com/")
    .ToObservable();
```

The `ToObservable` approach assumes you have already called the `async` method and have a `Task` to convert.

Another approach is to call `StartAsync`. `StartAsync` also calls the `async` method immediately but supports cancellation: if a subscription is disposed of, the `async` method is canceled:

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response = Observable
    .StartAsync(token => client.GetAsync("http://www.example.com/", token));
```

Both `ToObservable` and `StartAsync` immediately start the asynchronous operation without waiting for a subscription. If you want to create an observable that only starts the operation when subscribed to, you can use `FromAsync` (which also supports cancellation just like `StartAsync`):

```
var client = new HttpClient();
IObservable<HttpResponseMessage> response = Observable
    .FromAsync(token => client.GetAsync("http://www.example.com/", token));
```

`FromAsync` is notably different than `ToObservable` and `StartAsync`. Both `ToObservable` and `StartAsync` return an observable for an `async` operation that has already started. `FromAsync` starts a new, independent `async` operation every time it is subscribed to.

Finally, you can use special overloads of `SelectMany` to start asynchronous operations for each event in a source stream as they arrive. `SelectMany` also supports cancellation.

The following example takes an existing event stream of URLs and then initiates a request as each URL arrives:

```
IObservable<string> urls = ...
var client = new HttpClient();
IObservable<HttpResponseMessage> responses = urls
    .SelectMany((url, token) => client.GetAsync(url, token));
```

Discussion

Reactive Extensions existed before the introduction of `async` but added these operators (and others) so that it could interoperate well with `async` code. I recommend that you use the operators described even though you can build the same functionality using other Rx operators.

See Also

Recipe 7.5 covers consuming observable streams with asynchronous code.

Recipe 7.7 covers using dataflow blocks (which can contain asynchronous code) as sources of observable streams.

7.7. Rx Observables and Dataflow Meshes

Problem

Part of your solution uses Rx observables, and part of your solution uses dataflow meshes, and you need them to communicate.

Rx observables and dataflow meshes each have their own uses, with some conceptual overlap; this recipe shows how easily they work together so you can use the best tool for each part of the job.

Solution

First, let's consider using a dataflow block as an input to an observable stream. The following code creates a buffer block (which does no processing) and creates an observable interface from that block by calling `AsObservable`:

```
var buffer = new BufferBlock<int>();
IObservable<int> integers = buffer.AsObservable();
integers.Subscribe(data => Trace.WriteLine(data),
    ex => Trace.WriteLine(ex),
    () => Trace.WriteLine("Done"));

buffer.Post(13);
```

Buffer blocks and observable streams can be completed normally or with error, and the `AsObservable` method will translate the block completion (or fault) into the completion of the observable stream. However, if the block faults with an exception, that exception will be wrapped in an `AggregateException` when it is passed to the observable stream. This is similar to how linked blocks propagate their faults.

It is only a little more complicated to take a mesh and treat it as a destination for an observable stream. The following code calls `AsObserver` to allow a block to subscribe to an observable stream:

```
IObservable<DateTimeOffset> ticks =
    Observable.Interval(TimeSpan.FromSeconds(1))
    .Timestamp()
    .Select(x => x.Timestamp)
    .Take(5);

var display = new ActionBlock<DateTimeOffset>(x => Trace.WriteLine(x));
ticks.Subscribe(display.AsObserver());

try
{
    display.Completion.Wait();
    Trace.WriteLine("Done.");
}
```

```
catch (Exception ex)
{
    Trace.WriteLine(ex);
}
```

Just as before, the completion of the observable stream is translated to the completion of the block, and any errors from the observable stream are translated to a fault of the block.

Discussion

Dataflow blocks and observable streams share a lot of conceptual ground. They both have data pass through them, and they both understand completion and faults. They were designed for different scenarios; TPL Dataflow is intended for a mixture of asynchronous and parallel programming, while Rx is intended for reactive programming. However, the conceptual overlap is compatible enough that they work very well and naturally together.

See Also

Recipe 7.5 covers consuming observable streams with asynchronous code.

Recipe 7.6 covers using asynchronous code within an observable stream.

CHAPTER 8
Collections

Using the proper collections is essential in concurrent applications. I'm not talking about the standard collections like `List<T>`; I assume you already know about those. The purpose of this chapter is to introduce newer collections that are specifically intended for concurrent or asynchronous use.

Immutable collections are collection instances that can never change. At first glance, this sounds completely useless; but they're actually very useful even in single-threaded, nonconcurrent applications. Read-only operations (such as enumeration) act directly on the immutable instance. Write operations (such as adding an item) return a new immutable instance instead of changing the existing instance. This is not as wasteful as it first sounds because most of the time immutable collections share most of their memory. Furthermore, immutable collections have the advantage of being implicitly safe to access from multiple threads; since they cannot change, they are threadsafe.

Immutable collections are in the `Microsoft.Bcl.Immutable` NuGet package.

At the time of this writing, immutable collections are new, but they should be considered for all new development unless you *need* a mutable instance. If you're not familiar with immutable collections, I recommend that you start with Recipe 8.1, even if you don't need a stack or queue, because I'll cover several common patterns that all immutable collections follow.

If you need to construct an immutable collection with lots of existing elements, there are special ways to do this efficiently; the example code in these recipes only add elements one at a time. The MSDN documentation has details on how to efficiently construct

immutable collections if you need to speed up your initialization. Table 8-1 details the platform availability of immutable collections.

Table 8-1. Immutable collections platform availability

Platform	ImmutableStack<T>, etc.
.NET 4.5	✓
.NET 4.0	✗
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✓
Windows Phone SL 7.1	✗
Silverlight 5	✗

Threadsafe collections are mutable collection instances that can be changed by multiple threads simultaneously. Threadsafe collections use a mixture of fine-grained locks and lock-free techniques to ensure that threads are blocked for a minimal amount of time (and usually are not blocked at all). For many threadsafe collections, enumeration of the collection actually creates a snapshot of the collection and then enumerates that snapshot. The key advantage of threadsafe collections is that they can be accessed safely from multiple threads, yet the operations will only block your code for a short time, if at all. Table 8-2 details the platform availability of threadsafe collections.

Table 8-2. Threadsafe collections platform availability

Platform	ConcurrentDictionary<TKey, TValue>, etc.
.NET 4.5	✓
.NET 4.0	✓
Mono iOS/Droid	✓
Windows Store	✓
Windows Phone Apps 8.1	✓
Windows Phone SL 8.0	✗
Windows Phone SL 7.1	✗
Silverlight 5	✗

Producer/consumer collections are mutable collection instances that are designed with a specific purpose in mind: to allow (possibly multiple) producers to push items to the collection while allowing (possibly multiple) consumers to pull items out of the collection. So they act as a bridge between producer code and consumer code, and also have an option to limit the number of items in the collection. Producer/consumer collections can either have a blocking or asynchronous API. For example, when the collection is

empty, a blocking producer/consumer collection will block the calling consumer thread until another item is added; but an asynchronous producer/consumer collection will allow the calling consumer thread to asynchronously wait until another item is added. Table 8-3 details the platform availability of producer/consumer collections.

`AsyncProducerConsumerQueue<T>` and `AsyncCollection<T>` are in the `Nito.AsyncEx` NuGet package. `BufferBlock<T>` is in the `Microsoft.Tpl.Dataflow` NuGet package.

Table 8-3. Producer/consumer collections platform availability

Platform	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
.NET 4.5	✓	✓	✓	✓
.NET 4.0	✓	✗	✓	✓
Mono iOS/Droid	✓	✓	✓	✓
Windows Store	✓	✓	✓	✓
Windows Phone Apps 8.1	✓	✓	✓	✓
Windows Phone SL 8.0	✗	✓	✓	✗
Windows Phone SL 7.1	✗	✗	✓	✗
Silverlight 5	✗	✗	✓	✗

There are a number of different producer/consumer collections used in the recipes in this chapter, and different producer/consumer collections have different advantages. Table 8-4 may be helpful in determining which one you should use.

Table 8-4. Producer/consumer collections

Feature	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
Queue semantics	✓	✓	✓	✓
Stack/bag semantics	✓	✗	✗	✓
Synchronous API	✓	✓	✓	✓
Asynchronous API	✗	✓	✓	✓

Feature	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>	AsyncCollection<T>
Mobile platform support	Partial	Partial	✓	Partial
Tested by Microsoft	✓	✓	✗	✗

8.1. Immutable Stacks and Queues

Problem

You need a stack or queue that does not change very often and can be accessed by multiple threads safely.

For example, a queue can be used as a sequence of operations to perform, and a stack can be used as a sequence of undo operations.

Solution

Immutable stacks and queues are the simplest immutable collections. They behave very similarly to the standard `Stack<T>` and `Queue<T>`. Performance-wise, immutable stacks and queues have the same time complexity as standard stacks and queues; however, in simple scenarios where the collections are updated frequently, the standard stack and queue are faster.

Stacks are a first-in, first-out data structure. The following code creates an empty immutable stack, pushes two items, enumerates the items, and then pops an item:

```
var stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
stack = stack.Push(7);

// Displays "7" followed by "13".
foreach (var item in stack)
    Trace.WriteLine(item);

int lastItem;
stack = stack.Pop(out lastItem);
// lastItem == 7
```

Note that we keep overwriting the local variable `stack` in the preceding example. Immutable collections follow a pattern where they return an updated collection; the original collection reference is unchanged. This means that once you have a reference to a particular immutable collection instance, it will never change; consider the following example:

```
var stack = ImmutableStack<int>.Empty;
stack = stack.Push(13);
var biggerStack = stack.Push(7);

// Displays "7" followed by "13".
foreach (var item in biggerStack)
    Trace.WriteLine(item);

// Only displays "13".
foreach (var item in stack)
    Trace.WriteLine(item);
```

Under the covers, the two stacks are actually sharing the memory used to contain the item 13. This kind of implementation is very efficient while also allowing you to easily snapshot the current state. Each immutable collection instance is naturally threadsafe, but this can also be used in single-threaded applications. In my experience, immutable collections are especially useful when the code is more functional or when you need to store a large number of snapshots and want them to share memory as much as possible.

Queues are similar to stacks, except they are a first-in, last-out data structure. The following code creates an empty immutable queue, enqueues two items, enumerates the items, and then dequeues an item:

```
var queue = ImmutableQueue<int>.Empty;
queue = queue.Enqueue(13);
queue = queue.Enqueue(7);

// Displays "13" followed by "7".
foreach (var item in queue)
    Trace.WriteLine(item);

int nextItem;
queue = queue.Dequeue(out nextItem);
// Displays "13"
Trace.WriteLine(nextItem);
```

Discussion

This recipe introduced the two simplest immutable collections, the stack and the queue. However, we covered several important design philosophies that are true for *all* immutable collections:

- An instance of an immutable collection never changes.
- Since it never changes, it is naturally threadsafe.
- When you call a modifying method on an immutable collection, the modified collection is returned.

Immutable collections are ideal for sharing state. However, they don't work as well as communication conduits. In particular, don't use an immutable queue to communicate between threads; producer/consumer queues work much better for that.

`ImmutableStack<T>` and `ImmutableQueue<T>` are in the `Microsoft.Bcl.Immutable` NuGet package.

See Also

Recipe 8.6 covers threadsafe (blocking) mutable queues.

Recipe 8.7 covers threadsafe (blocking) mutable stacks.

Recipe 8.8 covers async-compatible mutable queues.

Recipe 8.9 covers async-compatible mutable stacks.

Recipe 8.10 covers blocking/asynchronous mutable queues.

8.2. Immutable Lists

Problem

You need a data structure you can index into that does not change very often and can be accessed by multiple threads safely.

A list is a general-purpose data structure that can be used for all kinds of application state.

Solution

Immutable lists do allow indexing, but you need to be aware of the performance characteristics. They're not just a drop-in replacement for `List<T>`.

`ImmutableList<T>` does support similar methods as `List<T>`, as this example shows:

```
var list = ImmutableList<int>.Empty;
list = list.Insert(0, 13);
list = list.Insert(0, 7);

// Displays "7" followed by "13".
foreach (var item in list)
    Trace.WriteLine(item);

list = list.RemoveAt(1);
```

However, the immutable list is internally organized as a binary tree. This is done so that immutable list instances may maximize the amount of memory they share with other instances. As a result, there are performance differences between `ImmutableList<T>` and `List<T>` for some common operations (Table 8-5).

Table 8-5. Performance difference of immutable lists

Operation	List<T>	ImmutableList<T>
Add	amortized O(1)	O(log N)
Insert	O(N)	O(log N)
RemoveAt	O(N)	O(log N)
Item[index]	O(1)	O(log N)

Of particular note, the indexing operation for `ImmutableList<T>` is O(log N), not O(1) as you may expect. If you are replacing `List<T>` with `ImmutableList<T>` in existing code, you'll need to consider how your algorithms access items in the collection.

This means that you should use `foreach` instead of `for` whenever possible. A `foreach` loop over an `ImmutableList<T>` executes in O(N) time, while a `for` loop over the same collection executes in O(N * log N) time:

```
// The best way to iterate over an ImmutableList<T>
foreach (var item in list)
    Trace.WriteLine(item);

// This will also work, but it will be much slower.
for (int i = 0; i != list.Count; ++i)
    Trace.WriteLine(list[i]);
```

Discussion

`ImmutableList<T>` is a good general-purpose data structure, but because of its performance differences, you can't blindly replace all your `List<T>` uses with it. `List<T>` is commonly used by default—that is, it's the one you use unless you *need* a different collection. `ImmutableList<T>` isn't quite that ubiquitous; you'll need to consider the other immutable collections carefully and choose the one that makes the most sense for your situation.

`ImmutableList<T>` is in the `Microsoft.Bcl.Immutable` NuGet package.

See Also

Recipe 8.1 covers immutable stacks and queues, which are like lists that only allow certain elements to be accessed.

MSDN documentation on `ImmutableList<T>.Builder`, an efficient way to populate an immutable list.

8.3. Immutable Sets

Problem

You need a data structure that does not need to store duplicates, does not change very often, and can be accessed by multiple threads safely.

For example, an index of words from a file would be a good use case for a set.

Solution

There are two immutable set types: `ImmutableHashSet<T>` is just a collection of unique items, and `ImmutableSortedSet<T>` is a *sorted* collection of unique items. Both types of sets have a similar interface:

```
var hashSet = ImmutableHashSet<int>.Empty;
hashSet = hashSet.Add(13);
hashSet = hashSet.Add(7);

// Displays "7" and "13" in an unpredictable order.
foreach (var item in hashSet)
    Trace.WriteLine(item);

hashSet = hashSet.Remove(7);
```

Only the sorted set allows indexing into it like a list:

```
var sortedSet = ImmutableSortedSet<int>.Empty;
sortedSet = sortedSet.Add(13);
sortedSet = sortedSet.Add(7);

// Displays "7" followed by "13".
foreach (var item in hashSet)
    Trace.WriteLine(item);
var smallestItem = sortedSet[0];
// smallestItem == 7

sortedSet = sortedSet.Remove(7);
```

Unsorted sets and sorted sets have similar performance (see Table 8-6).

Table 8-6. Performance of immutable sets

Operation	ImmutableHashSet<T>	ImmutableSortedSet<T>
Add	O(log N)	O(log N)
Remove	O(log N)	O(log N)
Item[index]	n/a	O(log N)

However, I recommend you use an unsorted set unless you know it needs to be sorted. Many types only support basic equality and not full comparison, so an unsorted set can be used for many more types than a sorted set.

One important note about the sorted set is that its indexing is O(log N), not O(1), just like `ImmutableList<T>`, which we looked at in Recipe 8.2. This means the same caveat applies: you should use `foreach` instead of `for` whenever possible with an `ImmutableSortedSet<T>`.

Discussion

Immutable sets are useful data structures, but populating a large immutable set can be slow. Most immutable collections have special builders that can be used to construct them quickly in a mutable way and then convert them into an immutable collection. This is true for many immutable collections, but I've found them most useful for immutable sets.

`ImmutableHashSet<T>` and `ImmutableSortedSet<T>` are in the `Microsoft.Bcl.Immutable` NuGet package.

See Also

Recipe 8.7 covers threadsafe mutable bags, which are similar to sets.

Recipe 8.9 covers async-compatible mutable bags.

MSDN documentation on `ImmutableHashSet<T>.Builder`, an efficient way to populate an immutable hash set.

MSDN documentation on `ImmutableSortedSet<T>.Builder`, an efficient way to populate an immutable sorted set.

8.4. Immutable Dictionaries

Problem

You need a key/value collection that does not change very often and can be accessed by multiple threads safely.

For example, you may want to store reference data in a *lookup collection*; the reference data rarely changes but should be available to different threads.

Solution

There are two immutable dictionary types: `ImmutableDictionary<TKey, TValue>` and `ImmutableSortedDictionary<TKey, TValue>`. As you may be able to guess, `ImmutableSortedDictionary` ensures that its elements are sorted, while the items in `ImmutableDictionary` have an unpredictable order.

Both of these collection types have very similar members:

```
var dictionary = ImmutableDictionary<int, string>.Empty;
dictionary = dictionary.Add(10, "Ten");
dictionary = dictionary.Add(21, "Twenty-One");
dictionary = dictionary.SetItem(10, "Diez");

// Displays "10Diez" and "21Twenty-One" in an unpredictable order.
foreach (var item in dictionary)
    Trace.WriteLine(item.Key + item.Value);

var ten = dictionary[10];
// ten == "Diez"

dictionary = dictionary.Remove(21);
```

Note the use of `SetItem`. In a mutable dictionary, you could do something like `dictionary[key] = item`, but immutable dictionaries must return the updated immutable dictionary, so they use the `SetItem` method instead:

```
var sortedDictionary = ImmutableSortedDictionary<int, string>.Empty;
sortedDictionary = sortedDictionary.Add(10, "Ten");
sortedDictionary = sortedDictionary.Add(21, "Twenty-One");
sortedDictionary = sortedDictionary.SetItem(10, "Diez");

// Displays "10Diez" followed by "21Twenty-One".
foreach (var item in sortedDictionary)
    Trace.WriteLine(item.Key + item.Value);

var ten = sortedDictionary[10];
// ten == "Diez"

sortedDictionary = sortedDictionary.Remove(21);
```

Unsorted dictionaries and sorted dictionaries have similar performance, but I recommend you use an unordered dictionary unless you need your elements to be sorted (see Table 8-7). Unsorted dictionaries can be a little faster overall. Furthermore, unsorted dictionaries can be used with any key types, while sorted dictionaries require their key types to be fully comparable.

Table 8-7. Performance of immutable dictionaries

Operation	ImmutableDictionary<TK,TV>	ImmutableSortedDictionary<TK,TV>
Add	O(log N)	O(log N)
SetItem	O(log N)	O(log N)
Item[key]	O(log N)	O(log N)
Remove	O(log N)	O(log N)

Discussion

In my experience, dictionaries are a common and useful tool when dealing with application state. They can be used in any kind of key/value or lookup scenario.

Like other immutable collections, immutable dictionaries have a builder mechanism for efficient construction if the dictionary contains many elements. For example, if you load your initial reference data at startup, you should use the builder mechanism to construct the initial immutable dictionary. On the other hand, if your reference data is gradually built up during your application's execution, then using the regular immutable dictionary `Add` method is likely acceptable.

`ImmutableDictionary<TK, TV>` and `ImmutableSortedDictionary<TK, TV>` are in the `Microsoft.Bcl.Immutable` NuGet package.

See Also

Recipe 8.5 covers threadsafe mutable dictionaries.

MSDN documentation on `ImmutableDictionary<TK,TV>.Builder`, an efficient way to populate an immutable dictionary.

MSDN documentation on `ImmutableSortedDictionary<TK,TV>.Builder`, an efficient way to populate an immutable sorted dictionary.

8.5. Threadsafe Dictionaries

Problem

You have a key/value collection that you need to keep in sync, even though multiple threads are both reading from and writing to it.

For example, consider a simple in-memory cache.

Solution

The `ConcurrentDictionary<TKey, TValue>` type in the .NET framework is a true gem of data structures. It is threadsafe, using a mixture of fine-grained locks and lock-free techniques to ensure fast access in the vast majority of scenarios.

Its API does take a bit of getting used to. It is not at all like the standard `Dictionary<TKey, TValue>` type, since it must deal with concurrent access from multiple threads. However, once you learn the basics in this recipe, you'll find `ConcurrentDictionary<TKey, TValue>` to be one of the most useful collection types.

First, let's cover writing a value to the collection. To set the value of a key, you can use `AddOrUpdate` as such:

```
var dictionary = new ConcurrentDictionary<int, string>();
var newValue = dictionary.AddOrUpdate(0,
    key => "Zero",
    (key, oldValue) => "Zero");
```

`AddOrUpdate` is a bit complex because it must do several things, depending on the current contents of the concurrent dictionary. The first method argument is the key. The second argument is a delegate that transforms the key (in this case, `0`) into a value to be added to the dictionary (in this case, `"Zero"`). This delegate is only invoked if the key does not exist in the dictionary. The third argument is another delegate that transforms the key (`0`) and the old value into an updated value to be stored in the dictionary (`"Zero"`). This delegate is only invoked if the key does exist in the dictionary. `AddOrUpdate` returns the new value for that key (the same value that was returned by one of the delegates).

Now for the part that really bends your brain: in order for the concurrent dictionary to work properly, `AddOrUpdate` *might* have to invoke either (or both) delegates multiple times. This is very rare, but it *can* happen. So your delegates should be simple and fast and not cause side effects. This means that your delegate should only create the value; it should not change any other variables in your application. The same principle holds for all delegates you pass to methods on `ConcurrentDictionary<TKey, TValue>`.

That was the hard part because it had to deal with all the thread safety concerns. The rest of the API is easier.

In fact, there are several other ways to add values to a dictionary. One shortcut is to just use indexing syntax:

```
// Using the same "dictionary" as above.
// Adds (or updates) key 0 to have the value "Zero".
dictionary[0] = "Zero";
```

Indexing syntax is less powerful; it does not give you the ability to update a value based on the existing value. However, the syntax is simpler and works fine if you already have the value you want to store in the dictionary.

Let's look at how to read a value. This can be easily done via `TryGetValue`:

```
// Using the same "dictionary" as above.
string currentValue;
bool keyExists = dictionary.TryGetValue(0, out currentValue);
```

`TryGetValue` will return `true` and set the `out` value if the key was found in the dictionary. If the key was not found, `TryGetValue` will return `false`. You can also use indexing syntax to read values, but I find that much less useful because it will throw an exception if a key is not found. Keep in mind that a concurrent dictionary has multiple threads reading, updating, adding, and removing values; in many situations, it's difficult to know whether a key exists or not until you attempt to read it.

Removing values is just as easy as reading them:

```
// Using the same "dictionary" as above.
string removedValue;
bool keyExisted = dictionary.TryRemove(0, out removedValue);
```

`TryRemove` is almost identical to `TryGetValue`, except (of course) it removes the key/value pair if the key was found in the dictionary.

Discussion

I think `ConcurrentDictionary<TKey, TValue>` is awesome, mainly because of the incredibly powerful `AddOrUpdate` method. However, it doesn't fit the bill in every situation. `ConcurrentDictionary<TKey, TValue>` is best when you have multiple threads reading and writing to a shared collection. If the updates are not constant (if they're more rare), than `ImmutableDictionary<TKey, TValue>` may be a better choice.

`ConcurrentDictionary<TKey, TValue>` is best in a shared-data situation, where multiple threads share the same collection. If some threads only add elements and other threads only remove elements, you'd be better served by a producer/consumer collection.

`ConcurrentDictionary<TKey, TValue>` is not the only threadsafe collection. The BCL also provides `ConcurrentStack<T>`, `ConcurrentQueue<T>`, and `ConcurrentBag<T>`. However, those threadsafe collections are seldomly used by themselves; they are usually only used in the implementation of producer/consumer collections, which we will cover in the rest of this chapter.

See Also

Recipe 8.4 covers immutable dictionaries, which are ideal if the contents of the dictionary change very rarely.

8.6. Blocking Queues

Problem

You need a conduit to pass messages or data from one thread to another. For example, one thread could be loading data, which it pushes down the conduit as it loads; meanwhile, there are other threads on the receiving end of the conduit that receive the data and process it.

Solution

The .NET type `BlockingCollection<T>` was designed to be this kind of conduit. By default, `BlockingCollection<T>` is a blocking queue, providing first-in, first-out behavior.

A blocking queue needs to be shared by multiple threads, so it is usually defined as a private, read-only field:

```
private readonly BlockingCollection<int> _blockingQueue =
    new BlockingCollection<int>();
```

Usually, a thread will *either* add items to the collection *or* remove items from the collection, but not both. Threads that add items are called *producer threads*, and threads that remove items are called *consumer threads*.

Producer threads can add items by calling `Add`, and when the producer thread is finished (i.e., all items have been added), it can finish the collection by calling `CompleteAdding`. This notifies the collection that no more items will be added to it, and the collection can then inform its consumers that there are no more items.

Here's a simple example of a producer that adds two items and then marks the collection complete:

```
_blockingQueue.Add(7);
_blockingQueue.Add(13);
_blockingQueue.CompleteAdding();
```

Consumer threads usually run in a loop, waiting for the next item and then processing it. If you take the producer code and put it in a separate thread (e.g., via `Task.Run`), then you can consume those items like this:

```
// Displays "7" followed by "13".
foreach (var item in _blockingQueue.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

If you want to have multiple consumers, `GetConsumingEnumerable` can be called from multiple threads at the same time. However, each item is only passed to one of those threads. When the collection is completed, the enumerable completes.

When you use conduits like this, you do need to consider what happens if your producers run faster than your consumers, unless you are sure that your consumers will *always* run faster. If you're producing items faster than you can consume them, then you may need to throttle your queue. `BlockingCollection<T>` makes this easy; you can throttle the number of items by passing the appropriate value when you create it. This simple example limits the collection to a single item:

```
BlockingCollection<int> _blockingQueue = new BlockingCollection<int>(
    boundedCapacity: 1);
```

Now, the same producer code will behave differently, as noted by the comments:

```
// This Add completes immediately.
_blockingQueue.Add(7);

// This Add waits for the 7 to be removed before it adds the 13.
_blockingQueue.Add(13);

_blockingQueue.CompleteAdding();
```

Discussion

The preceding examples all use `GetConsumingEnumerable` for the consumer threads; this is the most common scenario. However, there is also a `Take` member that allows a consumer to just consume a single item rather than run a loop consuming all the items.

Blocking queues are great when you have a separate thread (such as a thread-pool thread) acting as a producer or consumer. They're not as great when you want to access the conduit asynchronously—for example, if a UI thread wants to act as a consumer. We'll look at asynchronous queues in Recipe 8.8.

Whenever you introduce a conduit like this into your application, consider switching to the TPL Dataflow library. A lot of the time, using TPL Dataflow is simpler than building your own conduits and background threads. In particular, `BufferBlock<T>` can act like a blocking queue. However, TPL Dataflow is not available on all platforms, so in some cases, blocking queues are the appropriate design choice.

If you need maximum cross-platform support, you could also use `AsyncProducerConsumerQueue<T>` from the AsyncEx library, which can act like a blocking queue. Table 8-8 outlines the platform support for blocking queues.

Table 8-8. Platform support for blocking queues

Platform	BlockingCollection<T>	BufferBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	✓	✓	✓
.NET 4.0	✓	✗	✓
Mono iOS/Droid	✓	✓	✓
Windows Store	✓	✓	✓
Windows Phone Apps 8.1	✓	✓	✓
Windows Phone SL 8.0	✗	✓	✓
Windows Phone SL 7.1	✗	✗	✓
Silverlight 5	✗	✗	✓

See Also

Recipe 8.7 covers blocking stacks and bags if you want a similar conduit without first-in-first-out semantics.

Recipe 8.8 covers queues that have asynchronous rather than blocking APIs.

Recipe 8.10 covers queues that have both asynchronous and blocking APIs.

8.7. Blocking Stacks and Bags

Problem

You need a conduit to pass messages or data from one thread to another, but you don't want (or need) the conduit to have first-in, first-out semantics.

Solution

The .NET type `BlockingCollection<T>` acts as a blocking queue by default, but it can also act like any kind of producer/consumer collection. `BlockingCollection<T>` is actually a wrapper around a threadsafe collection that implements `IProducerConsumerCollection<T>`.

So, you can create a `BlockingCollection<T>` with last-in-first-out (stack) semantics or unordered (bag) semantics as such:

```
BlockingCollection<int> _blockingStack = new BlockingCollection<int>(
    new ConcurrentStack<int>());
```

```
BlockingCollection<int> _blockingBag = new BlockingCollection<int>(
    new ConcurrentBag<int>());
```

It's important to keep in mind that there are now race conditions around the ordering of the items. If we let the same producer code execute before any consumer code, and then execute the consumer code after the producer code, then the order of the items will be exactly like a stack:

```
// Producer code
_blockingStack.Add(7);
_blockingStack.Add(13);
_blockingStack.CompleteAdding();

// Consumer code
// Displays "13" followed by "7".
foreach (var item in _blockingStack.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

However, when the producer code and consumer code are on different threads (which is the usual case), the consumer will always get the most recently added item next. For example, the producer could add `7`, the consumer could take `7`, the producer could add `13`, and the consumer could take `13`. The consumer does *not* wait for `CompleteAdding` to be called before it returns the first item.

Discussion

The same considerations around throttling that apply to blocking queues also apply to blocking stacks and bags. If your producers run faster than your consumers and you need to limit the memory usage of your blocking stack/bag, you can use throttling exactly like we discussed in Recipe 8.6.

This recipe uses `GetConsumingEnumerable` for the consumer code; this is the most common scenario. However, there is also a `Take` member that allows a consumer to just consume a single item rather than run a loop consuming all the items.

If you want to access shared stacks or bags asynchronously rather than by blocking (for example, having your UI thread act as a consumer), see Recipe 8.9.

See Also

Recipe 8.6 covers blocking queues, which are much more commonly used than blocking stacks or bags.

Recipe 8.9 covers asynchronous stacks and bags.

8.8. Asynchronous Queues

Problem

You need a conduit to pass messages or data from one part of code to another in a first-in, first-out manner.

For example, one piece of code could be loading data, which it pushes down the conduit as it loads; meanwhile, the UI thread is receiving the data and displaying it.

Solution

What you need is a queue with an asynchronous API. There is no type like this in the core .NET framework, but there are a couple of options available from NuGet.

The first option is to use `BufferBlock<T>` from the TPL Dataflow library. The following simple example shows how to declare a `BufferBlock<T>`, what the producer code looks like, and what the consumer code looks like:

```
BufferBlock<int> _asyncQueue = new BufferBlock<int>();

// Producer code
await _asyncQueue.SendAsync(7);
await _asyncQueue.SendAsync(13);
_asyncQueue.Complete();

// Consumer code
// Displays "7" followed by "13".
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.ReceiveAsync());
```

`BufferBlock<T>` also has built-in support for throttling. For full details, see Recipe 8.10.

The example consumer code uses `OutputAvailbleAsync`, which is really only useful if you have a single consumer. If you have multiple consumers, it is possible that `OutputAvailbleAsync` will return `true` for more than one consumer even though there is only one item. If the queue is completed, then `DequeueAsync` will throw `InvalidOperationException`. So if you have multiple consumers, the consumer code usually looks more like this:

```
while (true)
{
    int item;
    try
    {
        item = await _asyncQueue.ReceiveAsync();
    }
    catch (InvalidOperationException)
    {
```

```
            break;
        }
        Trace.WriteLine(item);
    }
```

If TPL Dataflow is available on your platform, I recommend the `BufferBlock<T>` solution. Unfortunately, TPL Dataflow is not available everywhere. If `BufferBlock<T>` isn't available, you can use the `AsyncProducerConsumerQueue<T>` type from the `Nito.AsyncEx` NuGet library. The API is similar to but not exactly the same as `BufferBlock<T>`:

```
AsyncProducerConsumerQueue<int> _asyncQueue
    = new AsyncProducerConsumerQueue<int>();

// Producer code
await _asyncQueue.EnqueueAsync(7);
await _asyncQueue.EnqueueAsync(13);
await _asyncQueue.CompleteAdding();

// Consumer code
// Displays "7" followed by "13".
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.DequeueAsync());
```

`AsyncProducerConsumerQueue<T>` has support for throttling, which is necessary if your producers may run faster than your consumers. Just construct the queue with the appropriate value:

```
AsyncProducerConsumerQueue<int> _asyncQueue
    = new AsyncProducerConsumerQueue<int>(maxCount: 1);
```

Now, the same producer code will asynchronously wait appropriately:

```
// This Enqueue completes immediately.
await _asyncQueue.EnqueueAsync(7);

// This Enqueue (asynchronously) waits for the 7 to be removed
// before it enqueues the 13.
await _asyncQueue.EnqueueAsync(13);

await _asyncQueue.CompleteAdding();
```

This consumer code also uses `OutputAvailableAsync`, and has the same problems as `BufferBlock<T>`. The `AsyncProducerConsumerQueue<T>` type provides a `TryDequeueAsync` member that helps avoid cumbersome consumer code. If you have multiple consumers, the consumer code usually looks more like this:

```
while (true)
{
    var dequeueResult = await _asyncQueue.TryDequeueAsync();
    if (!dequeueResult.Success)
        break;
```

```
        Trace.WriteLine(dequeueResult.Item);
    }
```

Discussion

I do recommend that you use `BufferBlock<T>` over `AsyncProducerConsumerQueue<T>`, simply because `BufferBlock<T>` has been much more thoroughly tested. However, `BufferBlock<T>` is not available on many platforms, most notably older platforms (see Table 8-9).

The `BufferBlock<T>` type is in the `Microsoft.Tpl.Dataflow` NuGet package. The `AsyncProducerConsumerQueue<T>` type is in the `Nito.AsyncEx` NuGet package.

Table 8-9. Platform support for asynchronous queues

Platform	BufferBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	✓	✓
.NET 4.0	✗	✓
Mono iOS/Droid	✓	✓
Windows Store	✓	✓
Windows Phone Apps 8.1	✓	✓
Windows Phone SL 8.0	✓	✓
Windows Phone SL 7.1	✗	✓
Silverlight 5	✗	✓

See Also

Recipe 8.6 covers producer/consumer queues with blocking semantics rather than asynchronous semantics.

Recipe 8.10 covers producer/consumer queues that have *both* blocking and asynchronous semantics.

Recipe 8.7 covers asynchronous stacks and bags if you want a similar conduit without first-in, first-out semantics.

8.9. Asynchronous Stacks and Bags

Problem

You need a conduit to pass messages or data from one part of code to another, but you don't want (or need) the conduit to have first-in, first-out semantics.

Solution

The `Nito.AsyncEx` library provides a type `AsyncCollection<T>`, which acts like an asynchronous queue by default, but it can also act like any kind of producer/consumer collection. `AsyncCollection<T>` is a wrapper around an `IProducerConsumerCollection<T>`. `AsyncCollection<T>` is the `async` equivalent of the .NET `BlockingCollection<T>` that we saw in Recipe 8.7.

`AsyncCollection<T>` supports last-in, first-out (stack) or unordered (bag) semantics, based on whatever collection you pass to its constructor:

```
AsyncCollection<int> _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>());
AsyncCollection<int> _asyncBag = new AsyncCollection<int>(
    new ConcurrentBag<int>());
```

Note that there is a race condition around the ordering of items in the stack. If all producers complete before consumers start, then the order of items is like a regular stack:

```
// Producer code
await _asyncStack.AddAsync(7);
await _asyncStack.AddAsync(13);
await _asyncStack.CompleteAddingAsync();

// Consumer code
// Displays "13" followed by "7".
while (await _asyncStack.OutputAvailableAsync())
    Trace.WriteLine(await _asyncStack.TakeAsync());
```

However, when both producers and consumers are executing concurrently (which is the usual case), the consumer will always get the most recently added item next. This will cause the collection as a whole to act not quite like a stack. Of course, the bag collection has no ordering at all.

`AsyncCollection<T>` has support for throttling, which is necessary if producers may add to the collection faster than the consumers can remove from it. Just construct the collection with the appropriate value:

```
AsyncCollection<int> _asyncStack = new AsyncCollection<int>(
    new ConcurrentStack<int>(), maxCount: 1);
```

Now the same producer code will asynchronously wait as needed:

```
// This Add completes immediately.
await _asyncStack.AddAsync(7);

// This Add (asynchronously) waits for the 7 to be removed
// before it enqueues the 13.
await _asyncStack.AddAsync(13);

await _asyncStack.CompleteAddingAsync();
```

The example consumer code uses `OutputAvailbleAsync`, which has the same limitation as described in Recipe 8.8. If you have multiple consumers, the consumer code usually looks more like this:

```
while (true)
{
    var takeResult = await _asyncStack.TryTakeAsync();
    if (!takeResult.Success)
        break;
    Trace.WriteLine(takeResult.Item);
}
```

Discussion

`AsyncCollection<T>` is really just the asynchronous equivalent of `BlockingCollection<T>` and only supports the same platforms (see Table 8-10).

The `AsyncCollection<T>` type is in the `Nito.AsyncEx` NuGet package.

Table 8-10. Platform support for stacks and bags

Platform	BlockingCollection<T> (blocking)	AsyncCollection<T> (asynchronous)
.NET 4.5	✓	✓
.NET 4.0	✓	✓
Mono iOS/Droid	✓	✓
Windows Store	✓	✓
Windows Phone Apps 8.1	✓	✓
Windows Phone SL 8.0	✗	✗
Windows Phone SL 7.1	✗	✗
Silverlight 5	✗	✗

See Also

Recipe 8.8 covers asynchronous queues, which are much more common than asynchronous stacks or bags.

Recipe 8.7 covers synchronous (blocking) stacks and bags.

8.10. Blocking/Asynchronous Queues

Problem

You need a conduit to pass messages or data from one part of code to another in a first-in, first-out manner, and you need the flexibility to treat either the producer end or the consumer end as synchronous or asynchronous.

For example, a background thread may be loading data and pushing it into the conduit, and you want this thread to synchronously block if the conduit is too full. At the same time, the UI thread is receiving data from the conduit, and you want this thread to asynchronously pull data from the conduit so the UI remains responsive.

Solution

We've looked at blocking queues in Recipe 8.6 and asynchronous queues in Recipe 8.8, but there are a few queue types that support both blocking and asynchronous APIs.

The first is `BufferBlock<T>` and `ActionBlock<T>` from the TPL Dataflow NuGet library. `BufferBlock<T>` can be easily used as an asynchronous producer/consumer queue (see Recipe 8.8 for more details):

```
BufferBlock<int> queue = new BufferBlock<int>();

// Producer code
await queue.SendAsync(7);
await queue.SendAsync(13);
queue.Complete();

// Consumer code for a single consumer
while (await queue.OutputAvailableAsync())
    Trace.WriteLine(await queue.ReceiveAsync());

// Consumer code for multiple consumers
while (true)
{
    int item;
    try
    {
        item = await queue.ReceiveAsync();
```

```
        }
        catch (InvalidOperationException)
        {
            break;
        }

        Trace.WriteLine(item);
    }
```

`BufferBlock<T>` also supports a synchronous API for both producers and consumers:

```
    BufferBlock<int> queue = new BufferBlock<int>();

    // Producer code
    queue.Post(7);
    queue.Post(13);
    queue.Complete();

    // Consumer code
    while (true)
    {
        int item;
        try
        {
            item = queue.Receive();
        }
        catch (InvalidOperationException)
        {
            break;
        }

        Trace.WriteLine(item);
    }
```

However, the consumer code using `BufferBlock<T>` is rather awkward, since it is not the "dataflow way." The TPL Dataflow library includes a number of blocks that can be linked together, allowing you to define a reactive mesh. In this case, a producer/consumer queue completing with a particular action can be defined using `ActionBlock<T>`:

```
    // Consumer code is passed to queue constructor
    ActionBlock<int> queue = new ActionBlock<int>(item => Trace.WriteLine(item));

    // Asynchronous producer code
    await queue.SendAsync(7);
    await queue.SendAsync(13);

    // Synchronous producer code
    queue.Post(7);
    queue.Post(13);
    queue.Complete();
```

If the TPL Dataflow library is not available on your desired platform(s), then there is an `AsyncProducerConsumerQueue<T>` type in `Nito.AsyncEx` that also supports both synchronous and asynchronous methods:

```
AsyncProducerConsumerQueue<int> queue = new AsyncProducerConsumerQueue<int>();

// Asynchronous producer code
await queue.EnqueueAsync(7);
await queue.EnqueueAsync(13);

// Synchronous producer code
queue.Enqueue(7);
queue.Enqueue(13);

queue.CompleteAdding();

// Asynchronous single consumer code
while (await queue.OutputAvailableAsync())
    Trace.WriteLine(await queue.DequeueAsync());

// Asynchronous multi-consumer code
while (true)
{
    var result = await queue.TryDequeueAsync();
    if (!result.Success)
        break;
    Trace.WriteLine(result.Item);
}

// Synchronous consumer code
foreach (var item in queue.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

Discussion

Even though `AsyncProducerConsumerQueue<T>` supports a wider range of platforms, I recommend using `BufferBlock<T>` or `ActionBlock<T>` if possible because the TPL Dataflow library has been more extensively tested than the `Nito.AsyncEx` library.

All of the TPL Dataflow blocks as well as `AsyncProducerConsumerQueue<T>` also support throttling by passing options to their constructors. Throttling is necessary when you have producers that push items faster than your consumers can consume them, which could cause your application to take up large amounts of memory. Platform support for synchronous/asynchronous queues is outlined in Table 8-11.

The `BufferBlock<T>` and `ActionBlock<T>` types are in the `Microsoft.Tpl.Dataflow` NuGet package. The `AsyncProducerConsumerQueue<T>` type is in the `Nito.AsyncEx` NuGet package.

Table 8-11. Platform support for synchronous/asynchronous queues

Platform	BufferBlock<T> and ActionBlock<T>	AsyncProducerConsumerQueue<T>
.NET 4.5	✓	✓
.NET 4.0	✗	✓
Mono iOS/Droid	✓	✓
Windows Store	✓	✓
Windows Phone Apps 8.1	✓	✓
Windows Phone SL 8.0	✓	✓
Windows Phone SL 7.1	✗	✓
Silverlight 5	✗	✓

See Also

Recipe 8.6 covers blocking producer/consumer queues.

Recipe 8.8 covers asynchronous producer/consumer queues.

Recipe 4.4 covers throttling dataflow blocks.

CHAPTER 9
Cancellation

The .NET 4.0 framework introduced exhaustive and well-designed cancellation support. This support is cooperative, which means that cancellation can be requested but not enforced on code. Since cancellation is cooperative, it is not possible to cancel code unless it is written to support cancellation. For this reason, I recommend supporting cancellation in as much of your own code as possible.

Cancellation is a type of signal, with two different sides: a source that triggers the cancellation, and a receiver that responds to the cancellation. In .NET, the source is `CancellationTokenSource` and the receiver is `CancellationToken`. The recipes in this chapter will cover both sides of cancellation in normal usage and describe how to use the cancellation support to interoperate with nonstandard forms of cancellation.

Cancellation is treated as a special kind of error. The convention is that canceled code will throw an exception of type `OperationCanceledException` (or a derived type, such as `TaskCanceledException`). This way the calling code knows that the cancellation was observed.

To indicate to calling code that your method supports cancellation, you should take a `CancellationToken` as a parameter. This parameter is usually the last parameter, unless your method also reports progress (Recipe 2.3). You can also consider providing an overload or default parameter value for consumers that do not require cancellation:

```
public void CancelableMethodWithOverload(CancellationToken cancellationToken)
{
    // code goes here
}

public void CancelableMethodWithOverload()
{
    CancelableMethodWithOverload(CancellationToken.None);
}
```

```
public void CancelableMethodWithDefault(
    CancellationToken cancellationToken = default(CancellationToken))
{
    // code goes here
}
```

`CancellationToken.None` is a special value that is equivalent to `default(CancellationToken)` and represents a cancellation token that will never be canceled. Consumers pass this value when they don't ever want the operation to be canceled.

9.1. Issuing Cancellation Requests

Problem

You have cancelable code (that takes a `CancellationToken`) and you need to cancel it.

Solution

The `CancellationTokenSource` type is the source for a `CancellationToken`. The `CancellationToken` only enables code to respond to cancellation requests; the `CancellationTokenSource` members allow code to request cancellation.

Each `CancellationTokenSource` is independent from every other `CancellationTokenSource` (unless you link them, which we will consider in Recipe 9.8). The `Token` property returns a `CancellationToken` for that source, and the `Cancel` method issues the actual cancellation request.

The following code illustrates creating a `CancellationTokenSource` and using `Token` and `Cancel`. This code uses an `async` method because it's easier to illustrate in a short code sample; the same `Token`/`Cancel` pair is used to cancel *all* kinds of code:

```
void IssueCancelRequest()
{
    var cts = new CancellationTokenSource();
    var task = CancelableMethodAsync(cts.Token);

    // At this point, the operation has been started.

    // Issue the cancellation request.
    cts.Cancel();
}
```

In the example code above, the `task` variable is ignored after it has started running; in real-world code, that task would probably be stored somewhere and awaited so that the end user is aware of the final result.

When you cancel code, there is almost always a race condition. The cancelable code may have been *just about to finish* when the cancel request is made, and if it doesn't

happen to check its cancellation token before finishing, it will actually complete successfully. In fact, when you cancel code, there are three possibilities: it may respond to the cancellation request (throwing `OperationCanceledException`), it may finish successfully, or it may finish with an error unrelated to the cancellation (throwing a different exception).

The following code is just like the last, except that it awaits the task, illustrating all three possible results:

```
async Task IssueCancelRequestAsync()
{
    var cts = new CancellationTokenSource();
    var task = CancelableMethodAsync(cts.Token);

    // At this point, the operation is happily running.

    // Issue the cancellation request.
    cts.Cancel();

    // (Asynchronously) wait for the operation to finish.
    try
    {
        await task;
        // If we get here, the operation completed successfully
        //  before the cancellation took effect.
    }
    catch (OperationCanceledException)
    {
        // If we get here, the operation was canceled before it completed.
    }
    catch (Exception)
    {
        // If we get here, the operation completed with an error
        //  before the cancellation took effect.
        throw;
    }
}
```

Normally, setting up the `CancellationTokenSource` and issuing the cancellation are in separate methods. Once you cancel a `CancellationTokenSource` instance, it is permanently canceled. If you need another source, you'll need to create another instance. The following code is a more realistic GUI-based example that uses one button to start an asynchronous operation and another button to cancel it. It also disables and enables the "start" and "cancel" buttons so that there can only be one operation at a time:

```
private CancellationTokenSource _cts;

private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    StartButton.IsEnabled = false;
    CancelButton.IsEnabled = true;
```

```
        try
        {
            _cts = new CancellationTokenSource();
            var token = _cts.Token;
            await Task.Delay(TimeSpan.FromSeconds(5), token);
            MessageBox.Show("Delay completed successfully.");
        }
        catch (OperationCanceledException)
        {
            MessageBox.Show("Delay was canceled.");
        }
        catch (Exception)
        {
            MessageBox.Show("Delay completed with error.");
            throw;
        }
        finally
        {
            StartButton.IsEnabled = true;
            CancelButton.IsEnabled = false;
        }
    }

    private void CancelButton_Click(object sender, RoutedEventArgs e)
    {
        _cts.Cancel();
    }
```

Discussion

The most realistic example in this recipe used a GUI application, but don't get the impression that cancellation is just for user interfaces. Cancellation has its place on the server as well; for example, ASP.NET provides a cancellation token representing the request timeout. It's true that cancellation token *sources* are rarer on the server side, but there's no reason you *can't* use them; I have used a `CancellationTokenSource` to request cancellation when ASP.NET decides to unload the app domain.

See Also

Recipe 9.4 covers passing tokens to `async` code.

Recipe 9.5 covers passing tokens to parallel code.

Recipe 9.6 covers using tokens with reactive code.

Recipe 9.7 covers passing tokens to dataflow meshes.

9.2. Responding to Cancellation Requests by Polling

Problem

You have a loop in your code that needs to support cancellation.

Solution

When you have a processing loop in your code, then there isn't a lower-level API to which you can pass the `CancellationToken`. In this case, you should periodically check whether the token has been canceled. The following code observes the token periodically while executing a CPU-bound loop:

```
public int CancelableMethod(CancellationToken cancellationToken)
{
    for (int i = 0; i != 100; ++i)
    {
        Thread.Sleep(1000); // Some calculation goes here.
        cancellationToken.ThrowIfCancellationRequested();
    }
    return 42;
}
```

If your loop is very tight (i.e., if the body of your loop executes very quickly), then you may want to limit how often you check your cancellation token. As always, measure your performance before and after a change like this before deciding which way is best. The following code is similar to the previous example, but it has more iterations of a faster loop, so I added a limit to how often the token is checked:

```
public int CancelableMethod(CancellationToken cancellationToken)
{
    for (int i = 0; i != 100000; ++i)
    {
        Thread.Sleep(1); // Some calculation goes here.
        if (i % 1000 == 0)
            cancellationToken.ThrowIfCancellationRequested();
    }
    return 42;
}
```

The proper limit to use depends entirely on how much work you're doing and how responsive the cancellation needs to be.

Discussion

The majority of the time, your code should just pass through the `CancellationToken` to the next layer. We'll look at examples of this in Recipe 9.4, Recipe 9.5, Recipe 9.6, and Recipe 9.7. This polling recipe should only be used if you have a processing loop that needs to support cancellation.

There is another member on `CancellationToken` called `IsCancellationRequested`, which starts returning `true` when the token is canceled. Some people use this member to respond to cancellation, usually by returning a default or `null` value. However, I do not recommend that approach for most code. The standard cancellation pattern is to raise an `OperationCanceledException`, which is taken care of by `ThrowIfCancellationRequested`. If code further up the stack wants to catch the exception and act like the result is `null`, then that's fine, but any code taking a `CancellationToken` should follow the standard cancellation pattern. If you do decide not to follow the cancellation pattern, at least document it clearly.

`ThrowIfCancellationRequested` works by *polling* the cancellation token; your code has to call it at regular intervals. There is also a way to register a callback that is invoked when cancellation is requested. The callback approach is more about interoperating with other cancellation systems, so we'll cover that in Recipe 9.9.

See Also

Recipe 9.4 covers passing tokens to `async` code.

Recipe 9.5 covers passing tokens to parallel code.

Recipe 9.6 covers using tokens with reactive code.

Recipe 9.7 covers passing tokens to dataflow meshes.

Recipe 9.9 covers using callbacks instead of polling to respond to cancellation requests.

Recipe 9.1 covers issuing a cancellation request.

9.3. Canceling Due to Timeouts

Problem

You have some code that needs to stop running after a timeout.

Solution

Cancellation is a natural solution for timeout situations. A timeout is just one type of cancellation request. The code that needs to be canceled merely observes the cancellation token just like any other cancellation; it should neither know nor care that the cancellation source is a timer.

NET 4.5 introduces some convenience methos for cancellation token sources that automatically issue a cancel request based on a timer. You can pass the timeout into the constructor:

```
async Task IssueTimeoutAsync()
{
    var cts = new CancellationTokenSource(TimeSpan.FromSeconds(5));
    var token = cts.Token;
    await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

Alternatively, if you already have a `CancellationTokenSource` instance, you can start a timeout for that instance:

```
async Task IssueTimeoutAsync()
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    cts.CancelAfter(TimeSpan.FromSeconds(5));
    await Task.Delay(TimeSpan.FromSeconds(10), token);
}
```

The constructor is not available on .NET 4.0, but the `CancelAfter` method is provided by the `Microsoft.Bcl.Async` NuGet library for that platform.

Discussion

Whenever you need to execute code with a timeout, you should use `CancellationTokenSource` and `CancelAfter` (or the constructor). There are other ways to do the same thing, but using the existing cancellation system is the easiest and most efficient option.

Remember that the code to be canceled needs to observe the cancellation token; it is not possible to easily cancel uncancelable code.

See Also

Recipe 9.4 covers passing tokens to `async` code.

Recipe 9.5 covers passing tokens to parallel code.

Recipe 9.6 covers using tokens with reactive code.

Recipe 9.7 covers passing tokens to dataflow meshes.

9.4. Canceling async Code

Problem

You are using `async` code and need to support cancellation.

Solution

The easiest way to support cancellation in asynchronous code is to just pass the `CancellationToken` through to the next layer. This example code performs an asynchronous delay and then returns a value; it supports cancellation by just passing the token to `Task.Delay`:

```
public async Task<int> CancelableMethodAsync(CancellationToken cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(2), cancellationToken);
    return 42;
}
```

Many asynchronous APIs support `CancellationToken`, so enabling cancellation yourself is usually a simple matter of taking a token and passing it along. As a general rule, if your method calls APIs that take `CancellationToken`, then your method should also take a `CancellationToken` and pass it to every API that supports it.

Discussion

Unfortunately, there are some methods that do not support cancellation. When you are in this situation, there's no easy solution. It is not possible to safely stop arbitrary code unless it is wrapped in a separate executable. In this case, you do always have the option of *pretending* to cancel the operation by ignoring the result.

Cancellation should be provided as an option whenever possible. This is because proper cancellation at a higher level depends on proper cancellation at the lower level. So, when you are writing your own `async` methods, try your best to include support for cancellation; you never know what higher-level method will want to call yours, and it might need cancellation.

See Also

Recipe 9.1 covers issuing a cancellation request.

Recipe 9.3 covers using cancellation as a timeout.

9.5. Canceling Parallel Code

Problem

You are using parallel code and need to support cancellation.

Solution

The easiest way to support cancellation is to pass the `CancellationToken` through to the parallel code. `Parallel` methods support this by taking a `ParallelOptions` instance. You can set the `CancellationToken` on a `ParallelOptions` instance in the following manner:

```
static void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    Parallel.ForEach(matrices,
        new ParallelOptions { CancellationToken = token },
        matrix => matrix.Rotate(degrees));
}
```

Alternatively, it is possible to observe the `CancellationToken` directly in your loop body:

```
static void RotateMatrices2(IEnumerable<Matrix> matrices, float degrees,
    CancellationToken token)
{
    // Warning: not recommended; see below.
    Parallel.ForEach(matrices, matrix =>
    {
        matrix.Rotate(degrees);
        token.ThrowIfCancellationRequested();
    });
}
```

However, the alternative method is more work and does not compose as well; with the alternate method, the parallel loop will wrap the `OperationCanceledException` within an `AggregateException`. Also, if you pass the `CancellationToken` as part of a `ParallelOptions` instance, the `Parallel` class may make more intelligent decisions about how often to check the token. For these reasons, it is best to pass the token as an option.

Parallel LINQ (PLINQ) also has built-in support for cancellation, via the `WithCancellation` operator:

```
static IEnumerable<int> MultiplyBy2(IEnumerable<int> values,
    CancellationToken cancellationToken)
{
    return values.AsParallel()
        .WithCancellation(cancellationToken)
        .Select(item => item * 2);
}
```

Discussion

Supporting cancellation for parallel work is important for a good user experience. If your application is doing parallel work, it will use a large amount of CPU at least for a

short time. High CPU usage is something that users notice, even if it doesn't interfere with other applications on the same machine. So, I recommend supporting cancellation whenever you do parallel computation (or any other CPU-intensive work), even if the total time spent with high CPU usage is not extremely long.

See Also

Recipe 9.1 covers issuing a cancellation request.

9.6. Canceling Reactive Code

Problem

You have some reactive code, and you need it to be cancelable.

Solution

The Reactive Extensions library has a notion of a *subscription* to an observable stream. Your code can dispose of the subscription to unsubscribe from the stream. In many cases, this is sufficient to logically cancel the stream. For example, the following code subscribes to mouse clicks when one button is pressed and unsubscribes (cancels the subscription) when another button is pressed:

```
private IDisposable _mouseMovesSubscription;

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    var mouseMoves = Observable
        .FromEventPattern<MouseEventHandler, MouseEventArgs>(
            handler => (s, a) => handler(s, a),
            handler => MouseMove += handler,
            handler => MouseMove -= handler)
        .Select(x => x.EventArgs.GetPosition(this));
    _mouseMovesSubscription = mouseMoves.Subscribe(val =>
    {
        MousePositionLabel.Content = "(" + val.X + ", " + val.Y + ")";
    });
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    if (_mouseMovesSubscription != null)
        _mouseMovesSubscription.Dispose();
}
```

However, it can be really convenient to make Rx work with the `CancellationTokenSource`/`CancellationToken` system that everything else uses for cancellation. The rest of this recipe covers ways that Rx interacts with `CancellationToken`.

The first major use case is when the observable code is wrapped in asynchronous code. We considered this situation in Recipe 7.5, and now we want to add `CancellationToken` support. In general, the easiest way to do this is to perform all operations using reactive operators and then call `ToTask` to convert the last resulting element to an awaitable task. The following code shows how to asynchronously take the last element in a sequence:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int lastElement = await observable.TakeLast(1).ToTask(cancellationToken);
// or: int lastElement = await observable.ToTask(cancellationToken);
```

Taking the first element is very similar; we just modify the observable before calling `ToTask`:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
int firstElement = await observable.Take(1).ToTask(cancellationToken);
```

Asynchronously converting the entire observable sequence to a task is likewise similar:

```
CancellationToken cancellationToken = ...
IObservable<int> observable = ...
IList<int> allElements = await observable.ToList().ToTask(cancellationToken);
```

Finally, let's consider the reverse situation. We've looked at several ways to handle situations where Rx code responds to `CancellationToken`—that is, where a `CancellationTokenSource` cancel request is translated into an unsubscription (a `Dispose`). We can also go the other way: issuing a cancellation request as a response to disposal.

The `FromAsync`, `StartAsync`, and `SelectMany` operators all support cancellation, as we saw in Recipe 7.6. This covers the vast majority of use cases. Rx also provides a `CancellationDisposable` type, which we can use directly as such:

```
using (var cancellation = new CancellationDisposable())
{
    CancellationToken token = cancellation.Token;
    // Pass the token to methods that respond to it.
}
// At this point, the token is canceled.
```

Discussion

Rx has its own notion of cancellation: unsubscription. This recipe looked at several ways to make Rx play nicely with the universal cancellation framework introduced in .NET 4.0. As long as you are in the Rx world portion of your code, just use the Rx subscription/unsubscription system; it's cleanest if you only introduce `CancellationToken` support at the boundaries.

See Also

Recipe 7.5 covers asynchronous wrappers around Rx code (without cancellation support).

Recipe 7.6 covers Rx wrappers around asynchronous code (with cancellation support).

Recipe 9.1 covers issuing a cancellation request.

9.7. Canceling Dataflow Meshes

Problem

You are using dataflow meshes and need to support cancellation.

Solution

The best way to support cancellation in your own code is to pass the `CancellationToken` through to a cancelable API. Each block in a dataflow mesh supports cancellation as a part of its `DataflowBlockOptions`. If we want to extend our custom dataflow block with cancellation support, we just set the `CancellationToken` property on the block options:

```
IPropagatorBlock<int, int> CreateMyCustomBlock(
    CancellationToken cancellationToken)
{
    var blockOptions = new ExecutionDataflowBlockOptions
    {
        CancellationToken = cancellationToken
    };
    var multiplyBlock = new TransformBlock<int, int>(item => item * 2,
        blockOptions);
    var addBlock = new TransformBlock<int, int>(item => item + 2,
        blockOptions);
    var divideBlock = new TransformBlock<int, int>(item => item / 2,
        blockOptions);

    var flowCompletion = new DataflowLinkOptions
    {
        PropagateCompletion = true
    };
    multiplyBlock.LinkTo(addBlock, flowCompletion);
    addBlock.LinkTo(divideBlock, flowCompletion);

    return DataflowBlock.Encapsulate(multiplyBlock, divideBlock);
}
```

In the example, I applied the `CancellationToken` to every block in the mesh. This isn't strictly necessary. Since I'm also propagating completion along the links, I could just

apply it to the first block and allow it to propagate through. Cancellations are considered a special form of error, so the blocks further down the pipeline would be completed with an error as that error propagates through. However, if I am canceling a mesh, I may as well cancel every block simultaneously; so I usually just set the `CancellationToken` option on every block.

Discussion

In dataflow meshes, cancellation is *not* a form of flush. When a block is canceled, it drops all its input and refuses to take any new items. So if you cancel a block while it's running, you *will* lose data.

See Also

Recipe 9.1 covers issuing a cancellation request.

9.8. Injecting Cancellation Requests

Problem

You have a layer of your code that needs to respond to cancellation requests and also issue its own cancellation requests to the next layer.

Solution

The .NET 4.0 cancellation system has built-in support for this scenario, known as *linked cancellation tokens*. A cancellation token source can be created linked to one (or many) existing tokens. When you create a linked cancellation token source, the resulting token is canceled when any of the existing tokens is canceled or when the linked source is explicitly canceled.

The following code performs an asynchronous HTTP request. The token passed into this method represents cancellation requested by the end user, and this method also applies a timeout to the request:

```
async Task<HttpResponseMessage> GetWithTimeoutAsync(string url,
    CancellationToken cancellationToken)
{
    var client = new HttpClient();

    using (var cts = CancellationTokenSource
        .CreateLinkedTokenSource(cancellationToken))
    {
        cts.CancelAfter(TimeSpan.FromSeconds(2));
        var combinedToken = cts.Token;
```

```
            return await client.GetAsync(url, combinedToken);
        }
    }
```

The resulting `combinedToken` is canceled when either the user cancels the existing `cancellationToken` or when the linked source is canceled by `CancelAfter`.

Discussion

Although our example only used a single `CancellationToken` source, the `CreateLinkedTokenSource` method can take any number of cancellation tokens as parameters. This allows you to create a single combined token from which you can implement your logical cancellation. For example, ASP.NET provides one token that represents the request timing out (`HttpRequest.TimedOutToken`) and another token that represents the user disconnecting (`HttpResponse.ClientDisconnectedToken`); handler code may create a linked token that responds to either of these cancellation requests.

One thing to keep in mind is the lifetime of the linked cancellation token source. Our previous example is the usual use case, where one or more cancellation tokens are passed into our method, which then links them together and passes them on as a combined token. Note that our example code is using the `using` statement, which ensures that the linked cancellation token source is disposed of when the operation is complete (and the combined token is no longer being used). Consider what would happen if we did not dispose of the linked cancellation token source: it is possible that this method may be called multiple times with the same (long-lived) existing token, in which case we would link a new token source each time the method is invoked. Even after the HTTP requests complete (and nothing is using the combined token), that linked source is still attached to the existing token. To prevent memory leaks like this, dispose of the linked cancellation token source when you no longer need the combined token.

See Also

Recipe 9.1 covers issuing cancellation requests in general.

Recipe 9.3 covers using cancellation as a timeout.

9.9. Interop with Other Cancellation Systems

Problem

You have some external or legacy code with its own notion of cancellation, and you want to control it using a standard `CancellationToken`.

Solution

The `CancellationToken` has two primary ways to respond to a cancellation request: polling (which we covered in Recipe 9.2) and callbacks (the subject of this recipe). Polling is normally used for CPU-bound code, such as data processing loops; callbacks are normally used in all other scenarios. You can register a callback for a token using the `CancellationToken.Register` method.

For example, let's say we're wrapping the `System.Net.NetworkInformation.Ping` type and we want to be able to cancel a ping. The `Ping` class already has a `Task`-based API but does not support `CancellationToken`. Instead, the `Ping` type has its own `SendAsyncCancel` method that we can use to cancel a ping. So, we register a callback that invokes that method, as follows:

```
async Task<PingReply> PingAsync(string hostNameOrAddress,
    CancellationToken cancellationToken)
{
    var ping = new Ping();
    using (cancellationToken.Register(() => ping.SendAsyncCancel()))
    {
        return await ping.SendPingAsync(hostNameOrAddress);
    }
}
```

Now, when a cancellation is requested, it will invoke the `SendAsyncCancel` method for us, canceling the `SendPingAsync` method.

Discussion

The `CancellationToken.Register` method can be used to interoperate with any kind of alternative cancellation system. However, do bear in mind that when a method takes a `CancellationToken`, a cancellation request should only cancel that one operation. Some alternative cancellation systems implement a cancel by closing some resource, which can cancel multiple operations; this kind of cancellation system does not map well to a `CancellationToken`. If you do decide to wrap that kind of cancellation in a `CancellationToken`, you should document its unusual cancellation semantics.

Keep in mind the lifetime of the callback registration. The `Register` method returns a disposable that should be disposed of when that callback is no longer needed. The preceding example code uses a `using` statement to clean up when the asynchronous operation completes. If we did not have that `using` statement, then each time we call that example code with the same (long-lived) `CancellationToken`, it would add another callback (which in turn keeps the `Ping` object alive). To avoid memory and resource leaks, dispose of the callback registration when you no longer need the callback.

See Also

Recipe 9.2 covers responding to a cancellation token by polling rather than callbacks.

Recipe 9.1 covers issuing cancellation requests in general.

CHAPTER 10
Functional-Friendly OOP

Modern programs require asynchronous programming; these days servers must scale better than ever, and end-user applications must be more reponsive than ever. Developers are finding that they must learn asynchronous programming, and as they explore this world, they find that it often clashes with the traditional object-oriented programming that they're accustomed to.

The core reason for this is because asynchronous programming is functional. By "functional," I don't mean, "it works"; I mean it's a functional style of programming instead of an procedural style of programming. A lot of developers learned basic functional programming in college and have hardly touched it since. If code like `(car (cdr '(3 5 7)))` gives you a chill as repressed memories come flooding back, then you may be in that category. But don't fear; modern asynchronous programming is not that hard once you get used to it.

The major breakthrough with `async` is that you can still think procedurally while programming asynchronously. This makes asynchronous methods easier to write and understand. However, under the covers, asynchronous code is still functional in nature, and this causes some problems when people try to force `async` methods into classical object-oriented designs. The recipes in this chapter deal with those friction points where asynchronous code clashes with object-oriented programming.

These friction points are especially noticeable when translating an existing OOP code base into an `async`-friendly code base.

10.1. Async Interfaces and Inheritance

Problem

You have a method in your interface or base class that you want to make asynchronous.

Solution

The key to understanding this problem and its solution is to realize that `async` is an implementation detail. It is not possible to mark interface methods or abstract methods as `async`. However, you can define a method with the same signature as an `async` method, just without the `async` keyword.

Remember that *types* are awaitable, not *methods*. You can `await` a `Task` returned by a method, whether or not that method is actually `async`. So, an interface or abstract method can just return a `Task` (or `Task<T>`), and the return value of that method is awaitable.

The following code defines an interface with an asynchronous method (without the `async` keyword), an implementation of that interface (with `async`), and an independent method that consumes a method of the interface (via `await`):

```
interface IMyAsyncInterface
{
    Task<int> CountBytesAsync(string url);
}

class MyAsyncClass : IMyAsyncInterface
{
    public async Task<int> CountBytesAsync(string url)
    {
        var client = new HttpClient();
        var bytes = await client.GetByteArrayAsync(url);
        return bytes.Length;
    }
}

static async Task UseMyInterfaceAsync(IMyAsyncInterface service)
{
    var result = await service.CountBytesAsync("http://www.example.com");
    Trace.WriteLine(result);
}
```

This same pattern works for abstract methods in base classes.

An asynchronous method signature only means that the implementation *may* be asynchronous. It is possible for the actual implementation to be synchronous if it has no real asynchronous work to do. For example, a test stub may implement the same interface (without `async`) by using something like `FromResult`:

```
class MyAsyncClassStub : IMyAsyncInterface
{
    public Task<int> CountBytesAsync(string url)
    {
        return Task.FromResult(13);
```

```
    }
}
```

Discussion

At the time of this writing (2014), `async` and `await` are still pretty new. As asynchronous methods become more common, asynchronous methods on interfaces and base classes will become more common as well. They're not that hard to work with if you keep in mind that it is the return type that is awaitable (not the method) and that an asynchronous method definition may be implemented either asynchronously or synchronously.

See Also

Recipe 2.2 covers returning a completed task, implementing an asynchronous method signature with synchrononus code.

10.2. Async Construction: Factories

Problem

You are coding a type that requires some asynchronous work to be done in its constructor.

Solution

Constructors cannot be `async`, nor can they use the `await` keyword. It would certainly be useful to `await` in a constructor, but this would change the C# language considerably.

One possibility is to have a constructor paired with an `async` initialization method, so the type could be used like this:

```
var instance = new MyAsyncClass();
await instance.InitializeAsync();
```

However, there are disadvantages to this approach. It can be easy to forget to call the `InitializeAsync` method, and the instance is not usable immediately after it was constructed.

A better solution is to make the type its own factory. The following type illustrates the asynchronous factory method pattern:

```
class MyAsyncClass
{
    private MyAsyncClass()
    {
    }

    private async Task<MyAsyncClass> InitializeAsync()
```

```
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return this;
    }

    public static Task<MyAsyncClass> CreateAsync()
    {
        var result = new MyAsyncClass();
        return result.InitializeAsync();
    }
}
```

The constructor and `InitializeAsync` method are `private` so that other code cannot possibly misuse them; the only way to create an instance is via the static `CreateAsync` factory method, and calling code cannot access the instance until after the initialization is complete.

Other code can create an instance like this:

```
var instance = await MyAsyncClass.CreateAsync();
```

Discussion

The primary advantage of this pattern is that there is no way that other code can get an uninitialized instance of `MyAsyncClass`. That's why I prefer this pattern over other approaches whenever I can use it.

Unfortunately, this approach does not work in some scenarios—in particular, when your code is using a Dependency Injection provider. As of this writing (2014), no major Dependency Injection or Inversion of Control library works with `async` code. There are a couple of alternatives that you can consider in this case.

If the instance you're creating is actually a shared resource, then you can use the asynchronous lazy type discussed in Recipe 13.1. Otherwise, you can use the asynchronous initialization pattern discussed in Recipe 10.3.

Here's an example of what *not* to do:

```
class MyAsyncClass
{
    public MyAsyncClass()
    {
        InitializeAsync();
    }

    // BAD CODE!!
    private async void InitializeAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

At first glance, this seems like a reasonable approach: you get a regular constructor that kicks off an asynchronous operation; however, there are several drawbacks that are due to the use of `async void`. The first problem is that when the constructor completes, the instance is still being asynchronously initialized, and there isn't an obvious way to determine when the asynchronous initialization has completed. The second problem is with error handling: any exceptions raised from `InitializeAsync` can't be caught by any `catch` clauses surrounding the object construction.

See Also

Recipe 10.3 covers the asynchronous initilization pattern, a way of doing asynchronous construction that works with Dependency Injection/Inversion of Control containers.

Recipe 13.1 covers asynchronous lazy initialization, which is a viable solution if the instance is conceptually a shared resource or service.

10.3. Async Construction: The Asynchronous Initialization Pattern

Problem

You are coding a type that requires some asynchronous work to be done in its constructor, but you cannot use the asynchronous factory pattern (Recipe 10.2) because the instance is created via reflection (e.g., a Dependency Injection/Inversion of Control library, data binding, `Activator.CreateInstance`, etc.).

Solution

When you have this scenario, you *have* to return an uninitialized instance, but you can mitigate this by applying a common pattern: the asynchronous initialization pattern. Every type that requires asynchronous initialization should define a property as such:

```
Task Initialization { get; }
```

I usually like to define this in a marker interface for types that require asynchronous initialization:

```
/// <summary>
/// Marks a type as requiring asynchronous initialization
/// and provides the result of that initialization.
/// </summary>
public interface IAsyncInitialization
{
    /// <summary>
    /// The result of the asynchronous initialization of this instance.
    /// </summary>
```

```
    Task Initialization { get; }
}
```

When you implement this pattern, you should start the initialization (and assign the `Initialization` property) in the constructor. The results of the asynchronous initialization (including any exceptions) are exposed via that `Initialization` property. Here's an example implementation of a simple type using asynchronous initialization:

```
class MyFundamentalType : IMyFundamentalType, IAsyncInitialization
{
    public MyFundamentalType()
    {
        Initialization = InitializeAsync();
    }

    public Task Initialization { get; private set; }

    private async Task InitializeAsync()
    {
        // Asynchronously initialize this instance.
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

If you are using a Dependency Injection/Inversion of Control library, you can create and initialize an instance of this type using code like this:

```
IMyFundamentalType instance = UltimateDIFactory.Create<IMyFundamentalType>();
var instanceAsyncInit = instance as IAsyncInitialization;
if (instanceAsyncInit != null)
    await instanceAsyncInit.Initialization;
```

We can extend this pattern to allow composition of types with asynchronous initialization. The following example defines another type that depends on an `IMyFundamentalType` that we defined above:

```
class MyComposedType : IMyComposedType, IAsyncInitialization
{
    private readonly IMyFundamentalType _fundamental;

    public MyComposedType(IMyFundamentalType fundamental)
    {
        _fundamental = fundamental;
        Initialization = InitializeAsync();
    }

    public Task Initialization { get; private set; }

    private async Task InitializeAsync()
    {
        // Asynchronously wait for the fundamental instance to initialize,
        //  if necessary.
```

```
        var fundamentalAsyncInit = _fundamental as IAsyncInitialization;
        if (fundamentalAsyncInit != null)
            await fundamentalAsyncInit.Initialization;

        // Do our own initialization (synchronous or asynchronous).
        ...
    }
}
```

The composed type waits for all of its components to initialize before it proceeds with its initialization. The rule to follow is that every component should be initialized by the end of `InitializeAsync`. This ensures that all dependent types are initialized as part of the composed initialization. Any exceptions from a component initialization are propagated to the composed type's initialization.

Discussion

If you can, I recommend using asynchronous factories (Recipe 10.2) or asynchronous lazy initialization (Recipe 13.1) instead of this solution. Those are the best approaches because you never expose an uninitialized instance. However, if your instances are created by Dependency Injection/Inversion of Control, data binding, etc., then you are forced to expose an uninitialized instance, and in that case I recommend using the asynchronous initialization pattern in this recipe.

Remember from when we looked at asynchronous interfaces (Recipe 10.1) that an asynchronous method signature only means that the method *may* be asynchronous. The `MyComposedType.InitializeAsync` code above is a good example of this: if the `IMyFundamentalType` instance does not also implement `IAsyncInitialization` and `MyComposedType` has no asynchronous initialization of its own, then its `InitializeAsync` method will actually complete synchronously.

The code for checking whether an instance implements `IAsyncInitialization` and initializing it is a bit awkward, and it becomes more so when you have a composed type that depends on a larger number of components. It's easy enough to create a helper method that can be used to simplify the code:

```
public static class AsyncInitialization
{
    static Task WhenAllInitializedAsync(params object[] instances)
    {
        return Task.WhenAll(instances
            .OfType<IAsyncInitialization>()
            .Select(x => x.Initialization));
    }
}
```

You can call `InitializeAllAsync` and pass in whatever instances you want initialized; the method will ignore any instances that do not implement `IAsyncInitialization`.

The initialization code for a composed type that depends on three injected instances can then look like this:

```
private async Task InitializeAsync()
{
   // Asynchronously wait for all 3 instances to initialize, if necessary.
   await AsyncInitialization.WhenAllInitializedAsync(_fundamental,
       _anotherType, _yetAnother);

   // Do our own initialization (synchronous or asynchronous).
   ...
}
```

See Also

Recipe 10.2 covers asynchronous factories, which are a way to do asynchronous construction without exposing uninitialized instances.

Recipe 13.1 covers asynchronous lazy initialization, which can be used if the instance is a shared resource or service.

Recipe 10.1 covers asynchronous interfaces.

10.4. Async Properties

Problem

You have a property that you want to make `async`. The property is not used in data binding.

Solution

This is a problem that often comes up when converting existing code to use `async`; in this situation, you have a property whose getter invokes a method that is now asynchronous. However, there is no such thing as an "asynchronous property." It's not possible to use the `async` keyword with a property, and that's a good thing. Property getters should return current values; they should not be kicking off background operations:

```
// What we think we want (does not compile).
public int Data
{
    async get
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return 13;
    }
}
```

When you find that your code wants an "asynchronous property," what your code really *needs* is something a little different. The solution depends on whether your property value needs to be evaluated once or multiple times; you have a choice between these semantics:

- A value that is asynchronously evaluated each time it is read
- A value that is asynchronously evaluated once and is cached for future access

If your "asynchronous property" needs to kick off a new (asynchronous) evaluation each time it is read, then it is not a *property*. It is actually a *method* in disguise. If you encountered this situation when converting synchronous code to asynchronous, then it's time to admit that the original design was actually incorrect; the property really should have been a method all along:

```
// As an asynchronous method.
public async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}
```

It is *possible* to return a `Task<int>` directly from a property, like this:

```
// As a Task-returning property.
// This API design is questionable.
public Task<int> Data
{
    get { return GetDataAsync(); }
}

private async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}
```

However, I do not recommend this approach. If every access to a property is going to kick off a new asynchronous operation, then that "property" should really be a method. The fact that it's an asynchronous method makes it clearer that a new asynchronous operation is initiated every time, so the API is not misleading. Recipe 10.3 and Recipe 10.6 do use task-returning properties, but those properties apply to the instance as a whole; they do not start a new asynchronous operation every time they are read.

The preceding solution covers the scenario where the property value is evaluated every time it is retrieved. The other scenario is that the "asynchronous property" should only kick off a single (asynchronous) evaluation and that the resulting value should be cached for future use. In this case, you can use asynchronous lazy initialization. We'll cover this

in detail in Recipe 13.1, but in the meantime, here's an example of what the code would look like:

```
// As a cached value.
public AsyncLazy<int> Data
{
    get { return _data; }
}

private readonly AsyncLazy<int> _data =
    new AsyncLazy<int>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(1));
        return 13;
    });
```

The code will only execute the asynchronous evaluation once and then return that same value to all callers. Calling code looks like this:

```
int value = await instance.Data;
```

In this case, the property syntax is appropriate since there is only one evaluation happening.

Discussion

One of the important questions to ask yourself is whether reading the property should start a new asynchronous operation; if the answer is "yes," then use an asynchronous *method* instead of a property. If the property should act as a lazy-evaluated cache, then use asynchronous intiailization (see Recipe 13.1). We didn't cover properties that are used in data binding; that scenario will be covered in Recipe 13.3.

When you're converting a synchronous property to an "asynchronous property," here's an example of what *not* to do:

```
private async Task<int> GetDataAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    return 13;
}

public int Data
{
    // BAD CODE!!
    get { return GetDataAsync().Result; }
}
```

You don't want to use `Result` or `Wait` to force asynchronous code to be synchronous. In GUI and ASP.NET platforms, such code can easily cause deadlocks. Even if you work around the deadlocks, you would still be exposing a misleading API: a property getter

(which should be a fast, synchronous operation) is actually a blocking operation. These problems with blocking are discussed in more detail in Chapter 1.

While we're on the subject of properties in `async` code, it's worth thinking about how state relates to asynchronous code. This is especially true if you're converting a synchronous code base to asynchronous. Consider any state that you expose in your API (e.g., via properties); for each piece of state, ask yourself: what is the current state of an object that has an asynchronous operation in progress? There's no right answer, but it's important to think about the semantics you want and to document them.

For example, consider `Stream.Position`, which represents the current offset of the stream pointer. With the synchronous API, when you call `Stream.Read` or `Stream.Write`, the actual reading/writing is done and `Stream.Position` is updated to reflect the new position before the `Read` or `Write` method returns. The semantics are clear for synchronous code.

Now, consider `Stream.ReadAsync` and `Stream.WriteAsync`: when should `Stream.Position` be updated? When the read/write operation is complete, or before it actually happens? If it's updated before the operation completes, is it updated synchronously by the time `ReadAsync`/`WriteAsync` returns, or could it happen shortly after that?

This is a great example of how a property that exposes state has perfectly clear semantics for synchronous code but no obviously correct semantics for asynchronous code. It's not the end of the world—you just need to think about your entire API when async-enabling your types and document the semantics you choose.

See Also

Recipe 13.1 covers asynchronous lazy initialization in detail.

Recipe 13.3 covers "asynchronous properties" that need to support data binding.

10.5. Async Events

Problem

You have an event that you need to use with handlers that might be `async`, and you need to detect whether the event handlers have completed. Note that this is a rare situation when raising an event; usually, when you raise an event, you don't care when the handlers complete.

Solution

It's not feasible to detect when `async void` handlers have returned, so you need some alternative way to detect when the asynchronous handlers have completed. The

Windows Store platform introduced a concept called *deferrals* that we can use to track asynchronous handlers. An asynchronous handler allocates a deferral before its first `await`, and later notifies the deferral when it is complete. Synchronous handlers do not need to use deferrals.

The `Nito.AsyncEx` library includes a type called a `DeferralManager`, which is used by the component raising the event. This deferral manager then permits event handlers to allocate deferrals and keeps track of when all the deferrals have completed.

For each of your events where you need to wait for the handlers to complete, you first extend your event arguments type as such:

```
public class MyEventArgs : EventArgs
{
  private readonly DeferralManager _deferrals = new DeferralManager();

  ... // Your own constructors and properties.

  public IDisposable GetDeferral()
  {
    return _deferrals.GetDeferral();
  }

  internal Task WaitForDeferralsAsync()
  {
    return _deferrals.SignalAndWaitAsync();
  }
}
```

When you're dealing with asynchronous event handlers, it's best to make your event arguments type threadsafe. The easiest way to do this is to make it immutable (i.e., have all its properties be read-only).

Then, each time you raise the event, you can (asynchronously) wait for all asynchronous event handlers to complete. The following code will return a completed task if there are no handlers; otherwise, it will create a new instance of your event arguments type, pass it to the handlers, and wait for any asynchronous handlers to complete:

```
public event EventHandler<MyEventArgs> MyEvent;

private Task RaiseMyEventAsync()
{
  var handler = MyEvent;
  if (handler == null)
    return Task.FromResult(0);

  var args = new MyEventArgs(...);
  handler(this, args);
  return args.WaitForDeferralsAsync();
}
```

Asynchronous event handlers can then use the deferral within a `using` block; the deferral notifies the deferral manager when it is disposed of:

```
async void AsyncHandler(object sender, MyEventArgs args)
{
    using (args.GetDeferral())
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
    }
}
```

This is slightly different than how Windows Store deferrals work. In the Windows Store API, each event that needs deferrals defines its own deferral type, and that deferral type has an explicit `Complete` method rather than being `IDisposable`.

Discussion

There are logically two different kinds of events used in .NET, with very different semantics. I call these *notification events* and *command events* to distinguish them; this is not official terminology, just some terms that I chose for clarity. A notification event is an event that is raised to notify other components of some situation. A notification is purely one-way; the sender of the event does not care whether there are any receivers of the event. With notifications, the sender and receiver can be entirely disconnected. Most events are notification events; one example is a button click.

In contrast, a command event is an event that is raised to implement some functionality on behalf of the sending component. Command events are not "events" in the true sense of the term, though they are often implemented as .NET events. The sender of a command must wait until the receiver handles it before moving on. If you use events to implement the Visitor pattern, then those are command events. Lifecycle events are also command events, so ASP.NET page lifecycle events and Windows Store events, such as `Application.Suspending` fall into this category. Any event that is actually an implementation is also a command event (e.g., `BackgroundWorker.DoWork`).

Notification events do not require any special code to allow asynchronous handlers; the event handlers can be `async void` and they work just fine. When the event sender raises the event, the asynchronous event handlers aren't completed immediately, but that doesn't matter because they're just notification events. So, if your event is a notification event, the grand total amount of work you need to do to support asynchronous handlers is: nothing.

Command events are a different story. When you have a command event, you need a way to detect when the handlers have completed. The preceding solution with deferrals should only be used for command events.

The `DeferralManager` type is in the `Nito.AsyncEx` NuGet package.

See Also

Chapter 2 covers the basics of asynchronous programming.

10.6. Async Disposal

Problem

You have a type that allows asynchronous operations but also needs to allow disposal of its resources.

Solution

There are a couple of options for dealing with existing operations when disposing of an instance: you can either treat the disposal as a cancellation request that is applied to all existing operations, or you can implement an actual *asynchronous completion*.

Treating disposal as a cancellation has a historic precedence on Windows; types such as file streams and sockets cancel any existing reads or writes when they are closed. We can do something very similar in .NET by defining our own private `CancellationTokenSource` and passing that token to our internal operations. With this code, `Dispose` will cancel the operations but will not wait for those operations to complete:

```
class MyClass : IDisposable
{
    private readonly CancellationTokenSource _disposeCts =
        new CancellationTokenSource();

    public async Task<int> CalculateValueAsync()
    {
        await Task.Delay(TimeSpan.FromSeconds(2), _disposeCts.Token);
        return 13;
    }

    public void Dispose()
    {
        _disposeCts.Cancel();
    }
}
```

The preceding code shows the basic pattern around `Dispose`. In a real-world app, we should also put in checks that the object is not already disposed of and also allow the user to supply her own `CancellationToken` (using the technique from Recipe 9.8):

```
public async Task<int> CalculateValueAsync(CancellationToken cancellationToken)
{
    using (var combinedCts = CancellationTokenSource
        .CreateLinkedTokenSource(cancellationToken, _disposeCts.Token))
    {
        await Task.Delay(TimeSpan.FromSeconds(2), combinedCts.Token);
        return 13;
    }
}
```

Calling code will have any existing operations canceled when `Dispose` is called:

```
async Task Test()
{
    Task<int> task;
    using (var resource = new MyClass())
    {
        task = CalculateValueAsync();
    }

    // Throws OperationCanceledException.
    var result = await task;
}
```

For some types, implementing `Dispose` as a cancellation request works just fine (e.g., `HttpClient` has these semantics). However, other types need to know when all the operations have completed. For these types, you need some kind of asynchronous completion.

Asynchronous completion is very similar to asynchronous initialization (see Recipe 10.3): there isn't much in the way of official guidance, so I'll describe one possible pattern, which is based on how TPL Dataflow blocks work. The important parts of asynchronous completion can be wrapped up in an interface:

```
/// <summary>
/// Marks a type as requiring asynchronous completion and provides
/// the result of that completion.
/// </summary>
interface IAsyncCompletion
{
    /// <summary>
    /// Starts the completion of this instance. This is conceptually similar
    /// to <see cref="IDisposable.Dispose"/>.
    /// After you call this method, you should not invoke any other members of
    /// this instance except <see cref="Completion"/>.
    /// </summary>
    void Complete();
```

```
        /// <summary>
        /// Gets the result of the completion of this instance.
        /// </summary>
        Task Completion { get; }
    }
```

The implementing type can use code like this:

```
    class MyClass : IAsyncCompletion
    {
        private readonly TaskCompletionSource<object> _completion =
            new TaskCompletionSource<object>();
        private Task _completing;

        public Task Completion
        {
            get { return _completion.Task; }
        }

        public void Complete()
        {
            if (_completing != null)
                return;
            _completing = CompleteAsync();
        }

        private async Task CompleteAsync()
        {
            try
            {
                ... // Asynchronously wait for any existing operations.
            }
            catch (Exception ex)
            {
                _completion.TrySetException(ex);
            }
            finally
            {
                _completion.TrySetResult(null);
            }
        }
    }
```

Calling code is not exactly elegant; we can't use the `using` statement because `Dispose` must be asynchronous. However, we can define a pair of helper methods that allow us to do something similar to `using`:

```
    static class AsyncHelpers
    {
        public static async Task Using<TResource>(Func<TResource> construct,
            Func<TResource, Task> process) where TResource : IAsyncCompletion
        {
            // Create the resource we're using.
```

```
    var resource = construct();

    // Use the resource, catching any exceptions.
    Exception exception = null;
    try
    {
        await process(resource);
    }
    catch (Exception ex)
    {
        exception = ex;
    }

    // Complete (logically dispose) the resource.
    resource.Complete();
    await resource.Completion;

    // Re-throw the process delegate exception if necessary.
    if (exception != null)
        ExceptionDispatchInfo.Capture(exception).Throw();
}

public static async Task<TResult> Using<TResource, TResult>(
    Func<TResource> construct, Func<TResource,
    Task<TResult>> process) where TResource : IAsyncCompletion
{
    // Create the resource we're using.
    var resource = construct();

    // Use the resource, catching any exceptions.
    Exception exception = null;
    TResult result = default(TResult);
    try
    {
        result = await process(resource);
    }
    catch (Exception ex)
    {
        exception = ex;
    }

    // Complete (logically dispose) the resource.
    resource.Complete();
    try
    {
        await resource.Completion;
    }
    catch
    {
        // Only allow exceptions from Completion if the process
        //  delegate did not throw an exception.
        if (exception == null)
```

```
                throw;
            }

            // Re-throw the process delegate exception if necessary.
            if (exception != null)
                ExceptionDispatchInfo.Capture(exception).Throw();

            return result;
        }
    }
```

The code uses `ExceptionDispatchInfo` to preserve the stack trace of the exception. Once these helpers are in place, calling code can use the `Using` method as such:

```
async Task Test()
{
    await AsyncHelpers.Using(() => new MyClass(), async resource =>
    {
        // Use resource.
    });
}
```

Discussion

Asynchronous completion is definitely more awkward than implementing Dispose as a cancellation request, and the more complex approach should only be used when you really need it. In fact, most of the time you can get away with not disposing anything at all, which is certainly the easiest approach because you don't have to do anything.

The asynchronous completion pattern described in this recipe is used by TPL Dataflow blocks and a handful of other types (e.g., `ConcurrentExclusiveSchedulerPair`). Dataflow blocks also have another type of completion request indicating that they should complete with an error (`IDataflowBlock.Fault(Exception)`). This may make sense for your types as well, so take the `IAsyncCompletion` in this recipe as one example of how you can implement asynchronous completion.

This recipe has two patterns for handling disposal; it is also possible to use *both* of them if you want. This would give your type the semantics of a clean shutdown if the client code uses `Complete` and `Completion` or a "cancel" if the client code uses `Dispose`.

See Also

Recipe 10.3 covers the asynchronous initialization pattern.

The MSDN documentation for TPL Dataflow covers Dataflow Block Completion and the clean shutdown semantics of TPL Dataflow blocks.

Recipe 9.8 covers linked cancellation tokens.

Recipe 10.1 covers asynchronous interfaces.

CHAPTER 11
Synchronization

When your application makes use of concurrency (as practically all .NET applications do), then you need to watch out for situations where one piece of code needs to update data while other code needs to access the same data. Whenever this happens, you need to *synchronize* access to the data. The recipes in this chapter cover the most common types used to synchronize access. However, if you use the other recipes in this book appropriately, you'll find that a lot of the more common synchronization is already done for you by the appropriate libraries. Before diving into the synchronization recipes, let's take a closer look at some common situations where synchronization may or may not be required.

The synchronization explanations in this section are slightly simplified, but the conclusions are all correct.

There are two major types of synchronization: *communication* and *data protection*. Communication is used when one piece of code needs to notify another piece of code of some condition (e.g., a new message has arrived). We'll cover communication more thoroughly in the actual recipes; the remainder of this intro will discuss data protection.

We need to use synchronization to protect shared data when *all three* of these conditions are true:

- Multiple pieces of code are running concurrently.
- These pieces are accessing (reading or writing) the same data.
- At least one piece of code is updating (writing) the data.

The reason for the first condition should be obvious; if your entire code just runs from top to bottom and nothing ever happens concurrently, then you never have to worry about synchronization. This is the case for some simple Console applications, but the vast majority of .NET applications do use *some* kind of concurrency. The second condition means that if each piece of code has its own local data that it doesn't *share*, then there's no need for synchronization; the local data is independent from any other pieces of code. There's also no need for synchronization if there is shared data but the data never changes; the third condition covers scenarios like configuration values and the like that are set at the beginning of the application and then never change. If the shared data is only read, then it doesn't need synchronization.

The purpose of data protection is to provide each piece of code with a consistent view of the data. If one piece of code is updating the data, then we use synchronization to make those updates appear atomic to the rest of the system.

It takes some practice to learn when synchronization is necessary, so we'll walk through a few examples before actually starting the recipes in this chapter. As our first example, consider the following code:

```
async Task MyMethodAsync()
{
    int val = 10;
    await Task.Delay(TimeSpan.FromSeconds(1));
    val = val + 1;
    await Task.Delay(TimeSpan.FromSeconds(1));
    val = val - 1;
    await Task.Delay(TimeSpan.FromSeconds(1));
    Trace.WriteLine(val);
}
```

If this method is called from a thread-pool thread (e.g., from within `Task.Run`), then the lines of code accessing `val` may run on separate thread-pool threads. But does it need synchronization? No, because none of them can be running at the same time. The method is asynchronous, but it is also sequential (meaning it progresses one part at a time).

OK, let's complicate the example a bit. This time we'll run concurrent asynchronous code:

```
class SharedData
{
    public int Value { get; set; }
}

async Task ModifyValueAsync(SharedData data)
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    data.Value = data.Value + 1;
}
```

```
// WARNING: may require synchronization; see discussion below.
async Task<int> ModifyValueConcurrentlyAsync()
{
    var data = new SharedData();

    // Start three concurrent modifications.
    var task1 = ModifyValueAsync(data);
    var task2 = ModifyValueAsync(data);
    var task3 = ModifyValueAsync(data);

    await Task.WhenAll(task1, task2, task3);
    return data.Value;
}
```

In this case, we're starting three modifications that run concurrently. Do we need synchronization? The answer is, "it depends." If we know that the method is called from a GUI or ASP.NET context (or any context that only allows one piece of code to run at a time), then there is no synchronization necessary because when the actual `data` modification code runs, it runs at a different time than the other two `data` modifications. For example, if this is run in a GUI context, there is only one UI thread that will execute each of the `data` modifications, so it *must* do them one at a time. So, if we know the context is a one-at-a-time context, then there is no synchronization needed. However, if this same method is called from a thread-pool thread (e.g., from `Task.Run`), then synchronization *would* be necessary. In that case, the three `data` modifications could run on separate thread-pool threads and update `data.Value` simultaneously, so we would need to synchronize access to `data.Value`.

Now let's make our data a private field instead of something we pass around, and consider one more wrinkle:

```
private int value;

async Task ModifyValueAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    value = value + 1;
}

// WARNING: may require synchronization; see discussion below.
async Task<int> ModifyValueConcurrentlyAsync()
{
    // Start three concurrent modifications.
    var task1 = ModifyValueAsync();
    var task2 = ModifyValueAsync();
    var task3 = ModifyValueAsync();

    await Task.WhenAll(task1, task2, task3);
```

```
    return value;
}
```

The same discussion also applies to this code; if the context may be a thread pool context, then synchronization is definitely necessary. But there's an additional wrinkle here. Previously, we created a `SharedData` instance that was shared among the three modifying methods; this time, the shared data is an actual private field. This means that if the calling code calls `ModifyValueConcurrentlyAsync` multiple times, each of those separate calls shares the same `value`. We may want to apply synchronization even in a one-at-a-time context if we want to avoid that kind of sharing. To put that another way, if we want to make it so that each call to `ModifyValueConcurrentlyAsync` waits until all previous calls have completed, then we will need to add synchronization. This is true even if the context ensures that only one thread is used for all the code (i.e., the UI thread). Synchronization in this scenario is actually a kind of *throttling* for asynchronous methods (see Recipe 11.2).

Let's look at one more `async` example. You can use `Task.Run` to do what I call "simple parallelism"—a basic kind of parallel processing that doesn't provide the efficiency and configurability that the true parallelism of `Parallel`/PLINQ does. The following code updates a shared value using simple parallelism:

```
// BAD CODE!!
async Task<int> SimpleParallelismAsync()
{
    int val = 0;
    var task1 = Task.Run(() => { val = val + 1; });
    var task2 = Task.Run(() => { val = val + 1; });
    var task3 = Task.Run(() => { val = val + 1; });
    await Task.WhenAll(task1, task2, task3);
    return val;
}
```

In this case, we have three separate tasks running on the thread pool (via `Task.Run`), all modifying the same `val`. So, our conditions apply, and we certainly do need synchronization here. Note that we do need synchronization even though `val` is a local variable; it is still *shared* between threads even though it is local to this one method.

Moving on to true parallel code, let's consider an example that uses the `Parallel` type:

```
void IndependentParallelism(IEnumerable<int> values)
{
    Parallel.ForEach(values, item => Trace.WriteLine(item));
}
```

Since this code uses `Parallel`, we must assume we're running on multiple threads. However, the body of the parallel loop (`item => Trace.WriteLine(item)`) only reads from its own data; there's no data sharing between threads here. The `Parallel` class divides the data among threads so that none of them has to share its data. Each thread

running its loop body is independent from all the other threads running the same loop body. So, no synchronization of this code is necessary.

Let's look at an aggregation example similar to the one covered in Recipe 3.2:

```
// BAD CODE!!
int ParallelSum(IEnumerable<int> values)
{
    int result = 0;
    Parallel.ForEach(source: values,
        localInit: () => 0,
        body: (item, state, localValue) => localValue + item,
        localFinally: localValue => { result += localValue; });
    return result;
}
```

In this example, we are again using multiple threads; this time, each thread starts with its local value initialized to 0 (`() => 0`), and for each input value processed by that thread, it adds the input value to its local value (`(item, state, localValue) => localValue + item`). Finally, all the local values are added to the return value (`localValue => { result += localValue; }`). The first two steps aren't problematic because there's nothing shared between threads; each thread's local and input values are independent from all other threads' local and input values. However, the final step is problematic; when each thread's local value is added to the return value, we have a situation where there is a shared variable (`result`) that is accessed by multiple threads and updated by all of them. So, we need to use synchronization in that final step (see Recipe 11.1).

The PLINQ, dataflow, and reactive libraries are very similar to the `Parallel` examples: as long as your code is just dealing with its own input, it doesn't have to worry about synchronization. I find that if I use these libraries appropriately, there is very little need for me to add synchronization to most of my code.

Lastly, let's discuss collections for a bit. Remember that the three conditions requiring synchronization are *multiple pieces of code*, *shared data*, and *data updates*.

Immutable types are naturally threadsafe because they *cannot* change; it's not possible to update an immutable collection, so no synchronization is necessary. For example, this code does not require synchronization because when each separate thread-pool thread pushes a value onto the stack, it is actually creating a new immutable stack with that value, leaving the original `stack` unchanged:

```
async Task<bool> PlayWithStackAsync()
{
    var stack = ImmutableStack<int>.Empty;

    var task1 = Task.Run(() => Trace.WriteLine(stack.Push(3).Peek()));
    var task2 = Task.Run(() => Trace.WriteLine(stack.Push(5).Peek()));
    var task3 = Task.Run(() => Trace.WriteLine(stack.Push(7).Peek()));
    await Task.WhenAll(task1, task2, task3);
```

```
    return stack.IsEmpty; // Always returns true.
}
```

However, when your code uses immutable collections, it's common to have a shared "root" variable that is not itself immutable. In that case, you *do* have to use synchronization. In the following code, each thread pushes a value onto the stack (creating a new immutable stack) and then updates the shared root variable. In this example, we *do* need synchronization to update the `stack` variable:

```
// BAD CODE!!
async Task<bool> PlayWithStackAsync()
{
    var stack = ImmutableStack<int>.Empty;

    var task1 = Task.Run(() => { stack = stack.Push(3); });
    var task2 = Task.Run(() => { stack = stack.Push(5); });
    var task3 = Task.Run(() => { stack = stack.Push(7); });
    await Task.WhenAll(task1, task2, task3);

    return stack.IsEmpty;
}
```

Threadsafe collections (e.g., `ConcurrentDictionary`) are quite different. Unlike immutable collections, threadsafe collections can be updated. However, they have all the synchronization they need built in, so you don't have to worry about it. If the following code updated a `Dictionary` instead of a `ConcurrentDictionary`, it would need synchronization; but since it is updating a `ConcurrentDictionary`, it does not need synchronization:

```
async Task<int> ThreadsafeCollectionsAsync()
{
    var dictionary = new ConcurrentDictionary<int, int>();

    var task1 = Task.Run(() => { dictionary.TryAdd(2, 3); });
    var task2 = Task.Run(() => { dictionary.TryAdd(3, 5); });
    var task3 = Task.Run(() => { dictionary.TryAdd(5, 7); });
    await Task.WhenAll(task1, task2, task3);

    return dictionary.Count; // Always returns 3.
}
```

11.1. Blocking Locks

Problem

You have some shared data and need to safely read and write it from multiple threads.

Solution

The best solution for this situation is to use the `lock` statement. When a thread enters a lock, it will prevent any other threads from entering that lock until the lock is released:

```
class MyClass
{
    // This lock protects the _value field.
    private readonly object _mutex = new object();

    private int _value;

    public void Increment()
    {
        lock (_mutex)
        {
            _value = _value + 1;
        }
    }
}
```

Discussion

There are many other kinds of locks in the .NET framework, such as `Monitor`, `SpinLock`, and `ReaderWriterLockSlim`. These lock types should almost never be used in most applications. In particular, it is natural for developers to jump to `ReaderWriterLockSlim` when there is no need for that level of complexity. The basic `lock` statement handles 99% of cases quite well.

There are four important guidlines when using locks:

- Restrict lock visibility.
- Document what the lock protects.
- Minimize code under lock.
- Never execute arbitrary code while holding a lock.

First, you should strive to restrict lock visibility. The object used in the `lock` statement should be a private field and never should be exposed to any method outside the class. There is usually at most one lock per type; if you have more than one, consider refactoring that type into separate types. You *can* lock on any reference type, but I prefer to have a field specifically for use with the `lock` statement, as in the last example. In particular, you should never `lock(this)` or lock on any instance of `Type` or `string`; these locks can cause deadlocks because they are accessible from other code.

Second, document what the lock protects. This is easy to overlook when initially writing the code but becomes more important as the code grows in complexity.

Third, do your best to minimize the code that is executed while holding a lock. One thing to watch for is blocking calls; it is not ideal to block at all while holding a lock.

Finally, do not ever call arbitrary code under lock. Arbitrary code can include raising events, invoking virtual methods, or invoking delegates. If you must execute arbitrary code, do so after the lock is released.

See Also

Recipe 11.2 covers `async`-compatible locks. The `lock` statement is not compatible with `await`.

Recipe 11.3 covers signaling between threads. The `lock` statement is intended to protect shared data, not send signals between threads.

Recipe 11.5 covers throttling, which is a generalization of locking. A lock can be thought of as throttling to one at a time.

11.2. Async Locks

Problem

You have some shared data and need to safely read and write it from multiple code blocks, which may be using `await`.

Solution

The .NET framework `SemaphoreSlim` type has been updated in version 4.5 to be compatible with `async`. It can be used as such:

```
class MyClass
{
    // This lock protects the _value field.
    private readonly SemaphoreSlim _mutex = new SemaphoreSlim(1);

    private int _value;

    public async Task DelayAndIncrementAsync()
    {
        await _mutex.WaitAsync();
        try
        {
            var oldValue = _value;
            await Task.Delay(TimeSpan.FromSeconds(oldValue));
            _value = oldValue + 1;
        }
        finally
        {
```

```
            _mutex.Release();
        }
    }
}
```

However, `SemaphoreSlim` can only be used this way on .NET 4.5 and other newer platforms. If you are on an older platform or writing a portable class library, you can use the `AsyncLock` type from the `Nito.AsyncEx` library:

```
class MyClass
{
    // This lock protects the _value field.
    private readonly AsyncLock _mutex = new AsyncLock();

    private int _value;

    public async Task DelayAndIncrementAsync()
    {
        using (await _mutex.LockAsync())
        {
            var oldValue = _value;
            await Task.Delay(TimeSpan.FromSeconds(oldValue));
            _value = oldValue + 1;
        }
    }
}
```

Discussion

The same guidelines from Recipe 11.1 also apply here, specifically:

- Restrict lock visibility.
- Document what the lock protects.
- Minimize code under lock.
- Never execute arbitrary code while holding a lock.

Keep your lock instances private; do not expose them outside the class. Be sure to clearly document (and carefully think through) exactly what a lock instance protects. Minimize code that is executed while holding a lock. In particular, do not call arbitrary code; this includes raising events, invoking virtual methods, and invoking delegates. Platform support for asynchronous locks is detailed in Table 11-1.

Table 11-1. Platform support for asynchronous locks

Platform	SemaphoreSlim.WaitAsync	AsyncLock
.NET 4.5	✓	✓
.NET 4.0	✗	✓
Mono iOS/Droid	✓	✓

Platform	SemaphoreSlim.WaitAsync	AsyncLock
Windows Store	✓	✓
Windows Phone Apps 8.1	✓	✓
Windows Phone SL 8.0	✓	✓
Windows Phone SL 7.1	✗	✓
Silverlight 5	✗	✓

The `AsyncLock` type is in the `Nito.AsyncEx` NuGet package.

See Also

Recipe 11.4 covers `async`-compatible signaling. Locks are intended to protect shared data, not act as signals.

Recipe 11.5 covers throttling, which is a generalization of locking. A lock can be thought of as throttling to one at a time.

11.3. Blocking Signals

Problem

You have to send a notification from one thread to another.

Solution

The most common and general-purpose cross-thread signal is `ManualResetEventSlim`. A manual-reset event can be in one of two states: signaled or unsignaled. Any thread may set the event to a signaled state or reset the event to an unsignaled state. A thread may also wait for the event to be signaled.

The following two methods are invoked by separate threads; one thread waits for a signal from the other:

```
class MyClass
{
    private readonly ManualResetEventSlim _initialized =
        new ManualResetEventSlim();

    private int _value;

    public int WaitForInitialization()
```

```
        {
            _initialized.Wait();
            return _value;
        }

        public void InitializeFromAnotherThread()
        {
            _value = 13;
            _initialized.Set();
        }
    }
```

Discussion

`ManualResetEventSlim` is a great general-purpose signal from one thread to another, but you should only use it when appropriate. If the "signal" is actually a *message* sending some piece of data across threads, then consider using a producer/consumer queue. On the other hand, if the signals are just used to coordinate access to shared data, then you should use a lock instead.

There are other thread synchronization signal types in the .NET framework that are less commonly used. If `ManualResetEventSlim` doesn't suit your needs, consider `AutoResetEvent`, `CountdownEvent`, or `Barrier`.

See Also

Recipe 8.6 covers blocking producer/consumer queues.

Recipe 11.1 covers blocking locks.

Recipe 11.4 covers `async`-compatible signals.

11.4. Async Signals

Problem

You need to send a notification from one part of the code to another, and the receiver of the notification must wait for it asynchronously.

Solution

If the notification only needs to be sent once, then you can use `TaskCompletionSource<T>` to send the notification asynchronously. The sending code calls `TrySetResult`, and the receiving code awaits its `Task` property:

```
class MyClass
{
    private readonly TaskCompletionSource<object> _initialized =
```

```
        new TaskCompletionSource<object>();

    private int _value1;
    private int _value2;

    public async Task<int> WaitForInitializationAsync()
    {
        await _initialized.Task;
        return _value1 + _value2;
    }

    public void Initialize()
    {
        _value1 = 13;
        _value2 = 17;
        _initialized.TrySetResult(null);
    }
}
```

The `TaskCompletionSource<T>` type can be used to asynchronously wait for any kind of situation—in this case, a notification from another part of the code. This works well if the signal is only sent once, but does not work as well if you need to turn the signal off as well as on.

The `Nito.AsyncEx` library contains a type `AsyncManualResetEvent`, which is an approximate equivalent of `ManualResetEvent` for asynchronous code. The following example is fabricated, but it shows how to use the `AsyncManualResetEvent` type:

```
class MyClass
{
    private readonly AsyncManualResetEvent _connected =
        new AsyncManualResetEvent();

    public async Task WaitForConnectedAsync()
    {
        await _connected.WaitAsync();
    }

    public void ConnectedChanged(bool connected)
    {
        if (connected)
            _connected.Set();
        else
            _connected.Reset();
    }
}
```

Discussion

Signals are a general-purpose notification mechanism. But if that "signal" is a *message*, used to send data from one piece of code to another, then consider using a producer/

consumer queue. Similarly, do not use general-purpose signals just to coordinate access to shared data; in that situation, use a lock.

The `AsyncManualResetEvent` type is in the `Nito.AsyncEx` NuGet package.

See Also

Recipe 8.8 covers asynchronous producer/consumer queues.

Recipe 11.2 covers asynchronous locks.

Recipe 11.3 covers blocking signals, which can be used for notifications across threads.

11.5. Throttling

Problem

You have highly concurrent code that is actually *too* concurrent, and you need some way to throttle the concurrency.

Code is too concurrent when parts of the application are unable to keep up with other parts, causing data items to build up and consume memory. In this scenario, throttling parts of the code can prevent memory issues.

Solution

The solution varies based on the type of concurrency your code is doing. These solutions all restrict concurrency to a specific value. Reactive Extensions has more powerful options, such as sliding time windows; throttling Rx is covered more thoroughly in Recipe 5.4.

Dataflow and parallel code all have built-in options for throttling concurrency:

```
IPropagatorBlock<int, int> DataflowMultiplyBy2()
{
    var options = new ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = 10
    };

    return new TransformBlock<int, int>(data => data * 2, options);
}

// Using Parallel LINQ (PLINQ)
```

```
IEnumerable<int> ParallelMultiplyBy2(IEnumerable<int> values)
{
    return values.AsParallel()
        .WithDegreeOfParallelism(10)
        .Select(item => item * 2);
}

// Using the Parallel class
void ParallelRotateMatrices(IEnumerable<Matrix> matrices, float degrees)
{
    var options = new ParallelOptions
    {
        MaxDegreeOfParallelism = 10
    };
    Parallel.ForEach(matrices, options, matrix => matrix.Rotate(degrees));
}
```

Concurrent asynchronous code can be throttled by using `SemaphoreSlim`:

```
async Task<string[]> DownloadUrlsAsync(IEnumerable<string> urls)
{
    var httpClient = new HttpClient();
    var semaphore = new SemaphoreSlim(10);
    var tasks = urls.Select(async url =>
    {
        await semaphore.WaitAsync();
        try
        {
            return await httpClient.GetStringAsync(url);
        }
        finally
        {
            semaphore.Release();
        }
    }).ToArray();
    return await Task.WhenAll(tasks);
}
```

Discussion

Throttling may be necessary when you find your code is using too many resources (for example, CPU or network connections). Bear in mind that end users usually have less powerful machines than developers, so it is better to throttle by a little too much than not enough.

See Also

Recipe 5.4 covers throttling for reactive code.

CHAPTER 12
Scheduling

When a piece of code executes, it has to run on some thread somewhere. A *scheduler* is an object that decides where a certain piece of code runs. There are a few different scheduler types in the .NET framework, and they're used with slight differences by parallel and dataflow code.

I recommend that you *not* specify a scheduler whenever possible; the defaults are usually correct. For example, the `await` operator in asynchronous code will automatically resume the method within the same context, unless you override this default, as described in Recipe 2.7. Similarly, reactive code has reasonable default contexts for raising its events, which you can override with `ObserveOn`, as described in Recipe 5.2

However, if you need other code to execute in a specific context (e.g., a UI thread context, or an ASP.NET request context), then you can use the scheduling recipes in this chapter to control the scheduling of your code.

12.1. Scheduling Work to the Thread Pool

Problem

You have a piece of code that you explicitly want to execute on a thread-pool thread.

Solution

The vast majority of the time, you'll want to use `Task.Run`, which is quite simple. The following code blocks a thread-pool thread for two seconds:

```
Task task = Task.Run(() =>
{
    Thread.Sleep(TimeSpan.FromSeconds(2));
});
```

`Task.Run` also understands return values and asynchronous lambdas just fine. The task returned by `Task.Run` in the following code will complete after two seconds with a result of 13:

```
Task<int> task = Task.Run(async () =>
{
    await Task.Delay(TimeSpan.FromSeconds(2));
    return 13;
});
```

`Task.Run` returns a `Task` (or `Task<T>`), which can be naturally consumed by asynchronous or reactive code.

Discussion

`Task.Run` is ideal for UI applications, when you have time-consuming work to do that cannot be done on the UI thread. For example, Recipe 7.4 uses `Task.Run` to push parallel processing to a thread-pool thread. However, do not use `Task.Run` on ASP.NET unless you are *absolutely* sure you know what you're doing. On ASP.NET, request handling code is already running on a thread-pool thread, so pushing it onto *another* thread-pool thread is usually counterproductive.

`Task.Run` is an effective replacement for `BackgroundWorker`, `Delegate.BeginInvoke`, and `ThreadPool.QueueUserWorkItem`. None of those should be used in new code; code using `Task.Run` is much easier to write correctly and maintain over time. Furthermore, `Task.Run` handles the vast majority of use cases for `Thread`, so most uses of `Thread` can also be replaced with `Task.Run` (with rare exceptions, such as Single-Thread Apartment threads).

Parallel and dataflow code executes on the thread pool by default, so there's usually no need to use `Task.Run` with code executed by the `Parallel`, Parallel LINQ, or TPL Dataflow libraries.

If you are doing dynamic parallelism, then use `Task.Factory.StartNew` instead of `Task.Run`. This is necessary because the `Task` returned by `Task.Run` has its default options configured for asynchronous use (i.e., to be consumed by asynchronous or reactive code). It does not support advanced concepts, such as parent/child tasks, which are more common in dynamic parallel code.

See Also

Recipe 7.6 covers consuming asynchronous code (such as the task returned from `Task.Run`) with reactive code.

Recipe 7.4 covers asynchronously waiting for parallel code, which is most easily done via `Task.Run`.

Recipe 3.4 covers dynamic parallelism, a scenario where you should use `Task.Factory.StartNew` instead of `Task.Run`.

12.2. Executing Code with a Task Scheduler

Problem

You have multiple pieces of code that you need to execute in a certain way. For example, you may need all the pieces of code to execute on the UI thread, or you may need to execute only a certain number at a time.

This recipe deals with how to define and construct a scheduler for those pieces of code. Actually applying that scheduler is the subject of the next two recipes.

Solution

There are quite a few different types in .NET that can handle scheduling; this recipe focuses on `TaskScheduler` because it is portable and relatively easy to use.

The simplest `TaskScheduler` is `TaskScheduler.Default`, which queues work to the thread pool. You will seldomly specify `TaskScheduler.Default` in your own code, but it is important to be aware of it, since it is the default for many scheduling scenarios. `Task.Run`, parallel, and dataflow code all use `TaskScheduler.Default`.

You can capture a specific *context* and later schedule work back to it by using `TaskScheduler.FromCurrentSynchronizationContext`, as follows:

```
TaskScheduler scheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

This creates a `TaskScheduler` that captures the current `SynchronizationContext` and schedules code onto that context. `SynchronizationContext` is a type that represents a general-purpose scheduling context. There are several different contexts in the .NET framework; most UI frameworks provide a `SynchronizationContext` that represents the UI thread, and ASP.NET provides a `SynchronziationContext` that represents the HTTP request context.

Another powerful type introduced in .NET 4.5 is the `ConcurrentExclusiveSchedulerPair`, which is actually *two* schedulers that are related to each other. The `ConcurrentScheduler` member is a scheduler that allows multiple tasks to execute at the same time, as long as no task is executing on the `ExclusiveScheduler`. The `ExclusiveScheduler` only executes code one task at a time, and only when there is no task already executing on the `ConcurrentScheduler`:

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair();
TaskScheduler concurrent = schedulerPair.ConcurrentScheduler;
TaskScheduler exclusive = schedulerPair.ExclusiveScheduler;
```

One common use for `ConcurrentExclusiveSchedulerPair` is to just use the `ExclusiveScheduler` to ensure only one task is executed at a time. Code that executes on the `ExclusiveScheduler` will run on the thread pool but will be restricted to executing exclusive of all other code using the same `ExclusiveScheduler` instance.

Another use for `ConcurrentExclusiveSchedulerPair` is as a throttling scheduler. You can create a `ConcurrentExclusiveSchedulerPair` that will limit its own concurrency. In this scenario, the `ExclusiveScheduler` is usually not used:

```
var schedulerPair = new ConcurrentExclusiveSchedulerPair(TaskScheduler.Default,
    maxConcurrencyLevel: 8);
TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
```

Note that this kind of throttling only throttles code while it is *executing*; it is quite different than the kind of logical throttling covered in Recipe 11.5. In particular, asynchronous code is not considered to be executing while it is awaiting an operation. The `ConcurrentScheduler` throttles executing code; other throttling, such as `SemaphoreSlim`, throttles at a higher level (i.e., an entire `async` method).

Discussion

You may have noticed that the last code example passed `TaskScheduler.Default` into the constructor for `ConcurrentExclusiveSchedulerPair`. This is because `ConcurrentExclusiveSchedulerPair` actually applies its concurrent/exclusive logic around an existing `TaskScheduler`.

This recipe introduces `TaskScheduler.FromCurrentSynchronizationContext`, which is useful to execute code on a captured context. It is also possible to use `SynchonizationContext` directly to execute code on that context; however, I do not recommend this approach. Whenever possible, use the `await` operator to resume on an implicitly captured context or use a `TaskScheduler` wrapper.

Do not ever use platform-specific types to execute code on a UI thread. WPF, Silverlight, iOS, and Android all provide the `Dispatcher` type, Windows Store uses the `CoreDispatcher`, and Windows Forms has the `ISynchronizeInvoke` interface (i.e., `Control.Invoke`). Do not use any of these types in new code; just pretend they don't exist. Using them will tie your code to a specific platform unnecessarily. `SynchronizationContext` is a general-purpose abstraction around these types.

Reactive Extensions introduces a more general scheduler abstraction: `IScheduler`. An Rx scheduler is capable of wrapping any other kind of scheduler; the `TaskPoolScheduler` will wrap any `TaskFactory` (which contains a `TaskScheduler`). The Rx team also defined an `IScheduler` implementation that can be manually controlled for testing. If you need to actually use a scheduler abstraction, I'd recommend using the `IScheduler` from Rx; it's well designed, well defined, and test friendly. However, most of the

time you don't need a scheduler abstraction, and earlier libraries, such as the Task Parallel Library and TPL Dataflow, only understand the `TaskScheduler` type.

See Also

Recipe 12.3 covers applying a `TaskScheduler` to parallel code.

Recipe 12.4 covers applying a `TaskScheduler` to dataflow code.

Recipe 11.5 covers higher-level logical throttling.

Recipe 5.2 covers Reactive Extensions schedulers for event streams.

Recipe 6.6 covers the Reactive Extensions test scheduler.

12.3. Scheduling Parallel Code

Problem

You need to control how the individual pieces of code are executed in parallel code.

Solution

Once you create an appropriate `TaskScheduler` instance (see Recipe 12.2), you can include it in the options that you pass to a `Parallel` method. The following code takes a sequence of sequences of matrices; it starts a bunch of parallel loops and wants to limit the *total* parallelism of all loops simultaneously, regardless of how many matrices are in each sequence:

```
void RotateMatrices(IEnumerable<IEnumerable<Matrix>> collections, float degrees)
{
    var schedulerPair = new ConcurrentExclusiveSchedulerPair(
        TaskScheduler.Default, maxConcurrencyLevel: 8);
    TaskScheduler scheduler = schedulerPair.ConcurrentScheduler;
    ParallelOptions options = new ParallelOptions { TaskScheduler = scheduler };
    Parallel.ForEach(collections, options,
        matrices => Parallel.ForEach(matrices, options,
            matrix => matrix.Rotate(degrees)));
}
```

Discussion

`Parallel.Invoke` also takes an instance of `ParallelOptions`, so you can pass a `TaskScheduler` to `Parallel.Invoke` the same way as `Parallel.ForEach`. If you are doing dynamic parallel code, you can pass `TaskScheduler` directly to `TaskFactory.StartNew` or `Task.ContinueWith`.

There is no way to pass a `TaskScheduler` to Parallel LINQ (PLINQ) code.

See Also

Recipe 12.2 covers common task schedulers and how to choose between them.

12.4. Dataflow Synchronization Using Schedulers

Problem

You need to control how the individual pieces of code are executed in dataflow code.

Solution

Once you create an appropriate `TaskScheduler` instance (see Recipe 12.2), you can include it in the options that you pass to a dataflow block. When called from the UI thread, the following code creates a dataflow mesh that multiples all of its input values by two (using the thread pool) and then appends the resulting values to the items of a list box (on the UI thread):

```
var options = new ExecutionDataflowBlockOptions
{
    TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext(),
};
var multiplyBlock = new TransformBlock<int, int>(item => item * 2);
var displayBlock = new ActionBlock<int>(
    result => ListBox.Items.Add(result), options);
multiplyBlock.LinkTo(displayBlock);
```

Discussion

Specifying a `TaskScheduler` is especially useful if you need to coordinate the actions of blocks in different parts of your dataflow mesh. For example, you can use the `ConcurrentExclusiveSchedulerPair.ExclusiveScheduler` to ensure that blocks A and C never execute code at the same time, while allowing block B to execute whenever it wants.

Keep in mind that synchronization by `TaskScheduler` only applies while the code is *executing*. For example, if you have an action block that runs asynchronous code and you apply an exclusive scheduler, the code is not considered running when it is awaiting.

You can specify a `TaskScheduler` for any kind of dataflow block. Even though a block may not execute your code (e.g., `BufferBlock<T>`), it still has housekeeping tasks that it needs to do, and it will use the provided `TaskScheduler` for all of its internal work.

See Also

Recipe 12.2 covers common task schedulers and how to choose between them.

CHAPTER 13
Scenarios

In this chapter, we'll take a look at a variety of types and techniques to address some common scenarios when writing concurrent programs. These kinds of scenarios could fill up another entire book, so I've selected just a few that I've found the most useful.

13.1. Initializing Shared Resources

Problem

You have a resource that is shared between multiple parts of your code. This resource needs to be initialized the first time it is accessed.

Solution

The .NET framework includes a type specifically for this purpose: `Lazy<T>`. You construct an instance of this type with a factory delegate that is used to initialize the instance. The instance is then made available via the `Value` property. The following code illustrates the `Lazy<T>` type:

```
static int _simpleValue;
static readonly Lazy<int> MySharedInteger = new Lazy<int>(() => _simpleValue++);

void UseSharedInteger()
{
    int sharedValue = MySharedInteger.Value;
}
```

No matter how many threads call `UseSharedInteger` simultaneously, the factory delegate is only executed once, and all threads wait for the same instance. Once it is created, the instance is cached and all future access to the `Value` property returns the same instance (in the preceding example, `MySharedInteger.Value` will always be `0`).

A very similar approach can be used if the initialization requires asynchronous work; in this case, we use a `Lazy<Task<T>>`:

```
static int _simpleValue;
static readonly Lazy<Task<int>> MySharedAsyncInteger =
    new Lazy<Task<int>>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(2)).ConfigureAwait(false);
        return _simpleValue++;
    });

async Task GetSharedIntegerAsync()
{
    int sharedValue = await MySharedAsyncInteger.Value;
}
```

In this example, the delegate returns a `Task<int>`, that is, an integer value determined asynchronously. No matter how many parts of the code call `Value` simultaneously, the `Task<int>` is only created once and returned to all callers. Each caller then has the option of (asynchronously) waiting until the task completes by passing the task to `await`.

This is an acceptable pattern, but there is one additional consideration. The asynchronous delegate may be executed on any thread that calls `Value`, and that delegate will execute within that context. If there are different thread types that may call `Value` (e.g., a UI thread and a thread-pool thread, or two different ASP.NET request threads), then it may be better to always execute the asynchronous delegate on a thread-pool thread. This is easy enough to do by just wrapping the factory delegate in a call to `Task.Run`:

```
static readonly Lazy<Task<int>> MySharedAsyncInteger = new Lazy<Task<int>>(
    () => Task.Run(
        async () =>
        {
            await Task.Delay(TimeSpan.FromSeconds(2));
            return _simpleValue++;
        }));
```

Discussion

The final code sample is a general code pattern for asynchronous lazy initialization. However, it's a bit awkward. The AsyncEx library includes an `AsyncLazy<T>` type that acts just like a `Lazy<Task<T>>` that executes its factory delegate on the thread pool. It can also be awaited directly, so the declaration and usage look like this:

```
private static readonly AsyncLazy<int> MySharedAsyncInteger =
    new AsyncLazy<int>(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(2));
        return _simpleValue++;
    });
```

```
public async Task UseSharedIntegerAsync()
{
    int sharedValue = await MySharedAsyncInteger;
}
```

The `AsyncLazy<T>` type is in the `Nito.AsyncEx` NuGet package.

See Also

Chapter 1 covers basic `async`/`await` programming.

Recipe 12.1 covers scheduling work to the thread pool.

13.2. Rx Deferred Evaluation

Problem

You want to create a new source observable whenever someone subscribes to it. For example, you want each subscription to represent a different request to a web service.

Solution

The Rx library has an operator `Observable.Defer`, which will execute a delegate each time the observable is subscribed to. This delegate acts as a factory that creates an observable. The following code uses `Defer` to call an asynchronous method every time someone subscribes to the observable:

```
static void Main(string[] args)
{
    var invokeServerObservable = Observable.Defer(
        () => GetValueAsync().ToObservable());
    invokeServerObservable.Subscribe(_ => { });
    invokeServerObservable.Subscribe(_ => { });

    Console.ReadKey();
}

static async Task<int> GetValueAsync()
{
    Console.WriteLine("Calling server...");
    await Task.Delay(TimeSpan.FromSeconds(2));
    Console.WriteLine("Returning result...");
    return 13;
}
```

If you execute this code, you should see output like this:

```
Calling server...
Calling server...
Returning result...
Returning result...
```

Discussion

Your own code usually doesn't subscribe to an observable more than once, but some Rx operators do under the covers. For example, the `Observable.While` operator will re-subscribe to a source sequence as long as its condition is true. `Defer` allows you to define an observable that is reevaluated every time a new subscription comes in. This is useful if you need to refresh or update the data for that observable.

See Also

Recipe 7.6 covers wrapping asynchronous methods in observables.

13.3. Asynchronous Data Binding

Problem

You are retrieving data asynchronously and need to data-bind the results (e.g., in the ViewModel of a Model-View-ViewModel design).

Solution

When a property is used in data binding, it must immediately and synchronously return some kind of result. If the actual value needs to be determined asynchronously, you can return a default result and later update the property with the correct value.

Keep in mind that asynchronous operations can usually end with failure as well as success. Since we are writing a ViewModel, we could use data binding to update the UI for an error condition as well.

The AsyncEx library has a type `NotifyTaskCompletion` that can be used for this:

```
class MyViewModel
{
    public MyViewModel()
    {
        MyValue = NotifyTaskCompletion.Create(CalculateMyValueAsync());
    }

    public INotifyTaskCompletion<int> MyValue { get; private set; }

    private async Task<int> CalculateMyValueAsync()
```

```
    {
        await Task.Delay(TimeSpan.FromSeconds(10));
        return 13;
    }
}
```

It is possible to data-bind to various properties on the `INotifyTaskCompletion<T>` property, as follows:

```
<Grid>
    <Label Content="Loading..."
        Visibility="{Binding MyValue.IsNotCompleted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
    <Label Content="{Binding MyValue.Result}"
        Visibility="{Binding MyValue.IsSuccessfullyCompleted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
    <Label Content="An error occurred" Foreground="Red"
        Visibility="{Binding MyValue.IsFaulted,
            Converter={StaticResource BooleanToVisibilityConverter}}"/>
</Grid>
```

Discussion

It's also possible to write your own data-binding wrapper instead of using the one from the AsyncEx library. This code gives the basic idea:

```
class BindableTask<T> : INotifyPropertyChanged
{
    private readonly Task<T> _task;

    public BindableTask(Task<T> task)
    {
        _task = task;
        var _ = WatchTaskAsync();
    }

    private async Task WatchTaskAsync()
    {
        try
        {
            await _task;
        }
        catch
        {
        }

        OnPropertyChanged("IsNotCompleted");
        OnPropertyChanged("IsSuccessfullyCompleted");
        OnPropertyChanged("IsFaulted");
        OnPropertyChanged("Result");
    }
```

```
        public bool IsNotCompleted { get { return !_task.IsCompleted; } }
        public bool IsSuccessfullyCompleted
        { get { return _task.Status == TaskStatus.RanToCompletion; } }
        public bool IsFaulted { get { return _task.IsFaulted; } }
        public T Result
        { get { return IsSuccessfullyCompleted ? _task.Result : default(T); } }

        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
                handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
```

Note that there is an empty `catch` clause on purpose: we specifically want to catch all exceptions and handle those conditions via data binding. Also, we explicitly do not want to use `ConfigureAwait(false)` because the `PropertyChanged` event should be raised on the UI thread.

The `NotifyTaskCompletion` type is in the `Nito.AsyncEx` NuGet package.

See Also

Chapter 1 covers basic `async`/`await` programming.

Recipe 2.7 covers using `ConfigureAwait`.

13.4. Implicit State

Problem

You have some state variables that need to be accessible at different points in your call stack. For example, you have a current operation identifier that you want to use for logging but that you don't want to add as a parameter to every method.

Solution

The best solution is to add parameters to your methods, store data as members of a class, or use dependency injection to provide data to the different parts of your code. However, there are some situations where that would overcomplicate the code.

The `CallContext` type in .NET provides `LogicalSetData` and `LogicalGetData` methods that allow you to give your state a name and place it on a logical "context." When you are done with that state, you can call `FreeNamedDataSlot` to remove it from the context. The following code shows how to use these methods to set an operation identifier that is later read by a logging method:

```
void DoLongOperation()
{
    var operationId = Guid.NewGuid();
    CallContext.LogicalSetData("OperationId", operationId);

    DoSomeStepOfOperation();

    CallContext.FreeNamedDataSlot("OperationId");
}

void DoSomeStepOfOperation()
{
    // Do some logging here.
    Trace.WriteLine("In operation: " +
        CallContext.LogicalGetData("OperationId"));
}
```

Discussion

The logical call context can be used with `async` methods, but only on .NET 4.5 and above. If you try to use it on .NET 4.0 with the `Microsoft.Bcl.Async` NuGet package, the code will compile but will not work correctly.

You should only store immutable data in the logical call context. If you need to update a data value in the logical call context, then you should overwrite the existing value with another call to `LogicalSetData`.

The logical call context is not extremely performant. I recommend you add parameters to your methods or store the data as members of a class instead of using the implicit logical call context, if at all possible.

If you are writing an ASP.NET application, consider using `HttpContext.Current.Items`, which does the same thing but is more performant than `CallContext`.

See Also

Chapter 1 covers basic `async`/`await` programming.

Chapter 8 covers several immutable collections if you need to store complex data as implicit state.

Index

A

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

B

C

D

E

N

O

P

Q

R

S

T

U

V

W

About the Author

Stephen Cleary is a Christian, husband, father, and developer who makes his home in beautiful Northern Michigan. He likes speaking and writing, but at the end of the day, he enjoys being just a regular developer. Since joining the professional ranks in 1998, Steve has acquired a great deal of experience ranging from ARM firmware to Azure. He has contributed to open source from the very beginning, starting with the Boost C++ libraries and releasing several libraries and utilities of his own.

Colophon

The animal on the cover of *Concurrency in C# Cookbook* is a common palm civet (*Paradoxurus hermaphroditus*), also known as an Asian palm civet. Despite their Latin name, palm civets (like all other mammals) have two distinct sexes and are not hermaphroditic. For the most part, palm civets are solitary creatures that only spend time with each other during the mating season. They are native to Southeast Asia and the Indonesian islands, and have recently been introduced to Japan and the Lesser Sunda Islands.

Palm civets are small, furry creatures that can grow up to 21 inches long and weigh 11 pounds. They often have white or grey markings, which can include a facemask that resembles a raccoon's. Unlike other civet species, their tails do not have rings. The palm civet's best defense against predators is the smelly secretion that it ejects from its anal scent gland when threatened. Olfactory marking also comes into play during mating, when males and females will use scent trails to locate each other in the forest.

Palm civets are nocturnal omnivores, and they play an important role in maintaining tropical forest biodiversity by dispersing fruit seeds. They especially enjoy drinking palm flower sap, which when fermented becomes toddy, a sweet liquor; this habit has earned them the nickname "toddy cat." In some parts of its range, especially Southern China, palm civets are hunted for bush meat. However, the biggest threat to the palm civet population is the capture of wild individuals for use in the kopi luwak process. Kopi luwak is coffee made from beans that have been digested and passed by a civet, and was traditionally prepared using droppings from wild animals. However, the growing popularity of the drink has lead to civets being captured and kept in tiny cages on giant farms where they endure a diet of only coffee beans and get no exercise or outdoor exposure. Tony Wild, the coffee executive responsible for introducing kopi luwak to the West, now opposes the practice on the grounds of animal welfare, and started a campaign called "Cut the Crap" to halt its use.

The cover image is from Lydekker's *Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.