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 As a consequence, I’ll minimize my coverage of outdated techniques in this book.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<

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:

```csharp
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 whatever 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:

```csharp
async Task DoSomethingAsync()
{
    int val = 13;
```
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 artificially wrapped in a TargetInvocationException or AggregateException:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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:

```csharp
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)`

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:

```csharp
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.

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:

```csharp
interface IObserver<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:

```csharp
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:

```csharp
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 IDisposa bile 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:

```csharp
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:

```csharp
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 Select Many), 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 simultaneously 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 recipes 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

<table>
<thead>
<tr>
<th>Platform</th>
<th>async</th>
<th>Parallel</th>
<th>Rx</th>
<th>Dataflow</th>
<th>Concurrent collections</th>
<th>Immutable collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET 4.5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>.NET 4.0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mono iOS/Droid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Windows Store</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Windows Phone Apps 8.1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Windows Phone SL 8.0</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
</tr>
<tr>
<td>Windows Phone SL 7.1</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Silverlight 5</td>
<td>✓</td>
<td>❌</td>
<td>✓</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>
Want to read more?

You can buy this book at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!
Use discount code: OPC10
All orders over $29.95 qualify for free shipping within the US.

It’s also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.