Chapter 14. Concurrency and Asynchrony

Most applications need to deal with more than one thing happening at a time (*concurrency*). In this chapter, we start with the essential prerequisites, namely the basics of threading and tasks, and then describe in detail the principles of asynchrony and C#’s asynchronous functions.

In Chapter 22, we revisit multithreading in greater detail, and in Chapter 23, we cover the related topic of parallel programming.

Introduction

Following are the most common concurrency scenarios:

Writing a responsive user interface
: In WPF, mobile, and Windows Forms applications, you must run time-consuming tasks concurrently with the code that runs your user interface to maintain responsiveness.

Allowing requests to process simultaneously
: On a server, client requests can arrive concurrently and so must be handled in parallel to maintain scalability. If you use ASP.NET Core or Web API, .NET Core does this for you automatically. However, you still need to be aware of shared state (for instance, the effect of using static variables for caching).

Parallel programming

Code that performs intensive calculations can execute faster on multicore/multiprocessor computers if the workload is divided between cores (Chapter 23 is dedicated to this).

Speculative execution

On multicore machines, you can sometimes improve performance by predicting something that might need to be done and then doing it ahead of time. LINQPad uses this technique to speed up the creation of new queries. A variation is to run a number of different algorithms in parallel that all solve the same task. Whichever one finishes first “wins”—this is effective when you can’t know ahead of time which algorithm will execute fastest.

The general mechanism by which a program can simultaneously execute code is called *multithreading*. Multithreading is supported by both the CLR and operating system and is a fundamental concept in concurrency. Understanding the basics of threading, and in particular, the effects of threads on *shared state*, is therefore essential.

Threading

A *thread* is an execution path that can proceed independently of others.

Each thread runs within an operating system process, which provides an isolated environment in which a program runs. With a *single-threaded* program, just one thread runs in the process’s isolated environment and so that thread has exclusive access to it. With a *multithreaded* program, multiple threads run in a single process, sharing the same execution environment (memory, in particular).

This, in part, is why multithreading is useful: one thread can fetch data in the background, for instance, while another thread displays the data as it arrives. This data is referred to as *shared state*.

Creating a Thread

A *client* program (Console, WPF, UWP, or Windows Forms) starts in a single thread that's created automatically by the OS (the "main" thread). Here it lives out its life as a single-threaded application, unless you do otherwise, by creating more threads (directly or indirectly).[1]

You can create and start a new thread by instantiating a `Thread` object and calling its `Start` method. The simplest constructor for `Thread` takes a `ThreadStart` delegate: a parameterless method indicating where execution should begin. Here's an example:

```
// NB: All samples in this chapter assume the following namespace
imports:
using System;
using System.Threading;

class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);          // Kick off a new thread
    t.Start();                               // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }
  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}

// Typical Output:
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

The main thread creates a new thread `t` on which it runs a method that repeatedly prints the character *y*. Simultaneously, the main thread repeatedly prints the character *x*, as shown in Figure 14-1. On a single-core computer, the operating system must allocate “slices” of time to each thread (typically 20 ms in Windows) to simulate concurrency, resulting in repeated blocks of *x* and *y*. On a multicore or multiprocessor machine, the two threads can genuinely execute in parallel (subject to competition by other active processes on the computer), although you still get repeated blocks of *x* and *y* in this example because of subtleties in the mechanism by which `Console` handles concurrent requests.

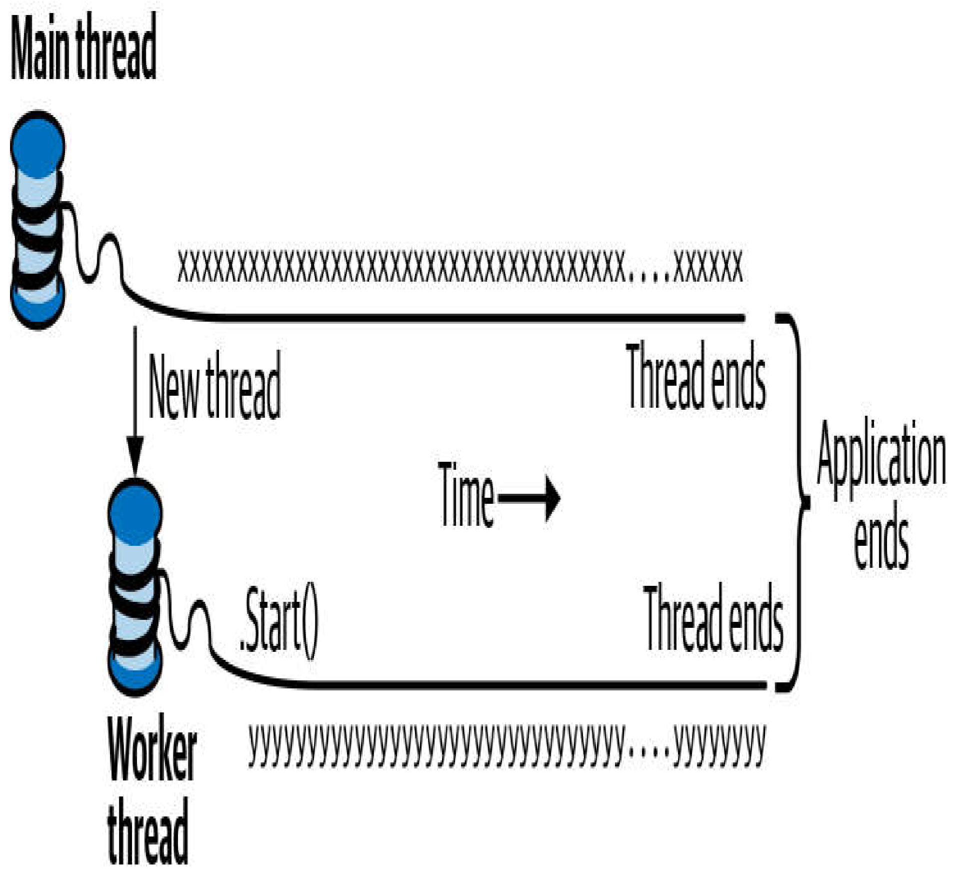
Main thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxx
New thread
Thread ends
Time
Application ends
.Start()
Thread ends
Worker thread
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyy

Figure 14-1. Starting a new thread

NOTE

A thread is said to be *preempted* at the points at which its execution is interspersed with the execution of code on another thread. The term often crops up in explaining why something has gone wrong!

After it's started, a thread's `IsAlive` property returns `true`, until the point at which the thread ends. A thread ends when the delegate passed to the `Thread`'s constructor finishes executing. After it's ended, a thread cannot restart.

Each thread has a `Name` property that you can set for the benefit of debugging. This is particularly useful in Visual Studio because the thread's name is displayed in the Threads Window and Debug Location toolbar. You can set a thread's name just once; attempts to change it later will throw an exception.

The static `Thread.CurrentThread` property gives you the currently executing thread:

```
Console.WriteLine (Thread.CurrentThread.Name);
```

Join and Sleep

You can wait for another thread to end by calling its `Join` method:

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

This prints "y" 1,000 times, followed by "Thread t has ended!" immediately afterward. You can include a timeout when calling `Join`, either in milliseconds or as a `TimeSpan`. It then returns `true` if the thread ended or `false` if it timed out.

`Thread.Sleep` pauses the current thread for a specified period:

```
Thread.Sleep (TimeSpan.FromHours (1));  // Sleep for 1 hour
Thread.Sleep (500);                     // Sleep for 500 milliseconds
```

`Thread.Sleep(0)` relinquishes the thread's current time slice immediately, voluntarily handing over the CPU to other threads. `Thread.Yield()` does the same thing except that it relinquishes only to threads running on the *same* processor.

NOTE

`Sleep(0)` or `Yield` is occasionally useful in production code for advanced performance tweaks. It's also an excellent diagnostic tool for helping to uncover thread safety issues: if inserting `Thread.Yield()` anywhere in your code breaks the program, you almost certainly have a bug.

While waiting on a `Sleep` or `Join`, a thread is blocked.

Blocking

A thread is deemed *blocked* when its execution is paused for some reason, such as when `Sleep`ing or waiting for another to end via `Join`. A blocked thread immediately *yields* its processor time slice, and from then on it consumes no processor time until its blocking condition is satisfied. You can test for a thread being blocked via its `ThreadState` property:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) !=
0;
```

NOTE

`ThreadState` is a flags enum, combining three “layers” of data in a bitwise fashion. Most values, however, are redundant, unused, or deprecated. The following extension method strips a `ThreadState` to one of four useful values: `Unstarted`, `Running`, `WaitSleepJoin`, and `Stopped`:

```
public static ThreadState Simplify (this ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

The `ThreadState` property is useful for diagnostic purposes but unsuitable for synchronization, because a thread’s state can change in between testing `ThreadState` and acting on that information.

When a thread blocks or unblocks, the OS performs a *context switch*. This incurs a small overhead, typically one or two microseconds.

I/O-BOUND VERSUS COMPUTE-BOUND

An operation that spends most of its time *waiting* for something to happen is called *I/O-bound*—an example is downloading a web page or calling `Console.ReadLine`. (I/O-bound operations typically involve input or output, but this is not a hard requirement: `Thread.Sleep` is also deemed I/O-bound.) In contrast, an operation that spends most of its time performing CPU-intensive work is called *compute-bound*.

BLOCKING VERSUS SPINNING

An I/O-bound operation works in one of two ways: it either waits *synchronously* on the current thread until the operation is complete (such as `Console.ReadLine`, `Thread.Sleep`, or `Thread.Join`), or operates *asynchronously*, firing a callback when the operation finishes some time thereafter (more on this later).

I/O-bound operations that wait synchronously spend most of their time blocking a thread. They can also "spin" in a loop periodically:

```
while (DateTime.Now < nextStartTime)
  Thread.Sleep (100);
```

Leaving aside that there are better ways to do this (such as timers or signaling constructs), another option is that a thread can spin continuously:

```
while (DateTime.Now < nextStartTime);
```

In general, this is very wasteful on processor time: as far as the CLR and OS are concerned, the thread is performing an important calculation and thus is allocated resources accordingly. In effect, we’ve turned what should be an I/O-bound operation into a compute-bound operation.

NOTE

There are a couple of nuances with regard to spinning versus blocking. First, spinning *very briefly* can be effective when you expect a condition to be satisfied soon (perhaps within a few microseconds) because it avoids the overhead and latency of a context switch. .NET Core provides special methods and classes to assist—see “SpinLock and SpinWait”.

Second, blocking does not incur a *zero* cost. This is because each thread ties up around 1 MB of memory for as long as it lives and causes an ongoing administrative overhead for the CLR and OS. For this reason, blocking can be troublesome in the context of heavily I/O-bound programs that need to handle hundreds or thousands of concurrent operations. Instead, such programs need to use a callback-based approach, rescinding their thread entirely while waiting. This is (in part) the purpose of the asynchronous patterns that we discuss later.

Local versus Shared State

The CLR assigns each thread its own memory stack so that local variables are kept separate. In the next example, we define a method with a local variable and then call the method simultaneously on the main thread and a newly created thread:

```
static void Main()
{
  new Thread (Go).Start();      // Call Go() on a new thread
  Go();                         // Call Go() on the main thread
}

static void Go()
{
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

A separate copy of the `cycles` variable is created on each thread's memory stack, and so the output is, predictably, 10 question marks.

Threads share data if they have a common reference to the same object instance:

```
class ThreadTest
{
  bool _done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest();   // Create a common instance
    new Thread (tt.Go).Start();
    tt.Go();
  }

  void Go()   // Note that this is an instance method
  {
     if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Because both threads call `Go()` on the same `ThreadTest` instance, they share the `_done` field. This results in "Done" being printed once instead of twice.

Local variables captured by a lambda expression or anonymous delegate are converted by the compiler into fields and so can also be shared:

```
static void Main()
{
  bool done = false;
  ThreadStart action = () =>
  {
    if (!done) { done = true; Console.WriteLine ("Done"); }
  };
  new Thread (action).Start();
  action();
}
```

Static fields offer another way to share data between threads:

```
class ThreadTest
{
  static bool _done;    // Static fields are shared between all threads
                        // in the same application domain.
  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

All three examples illustrate another key concept: that of thread safety (or rather, lack of it!). The output is actually indeterminate: it's possible (though unlikely) that "Done" could be printed twice. If, however, we swap the order of statements in the `Go` method, the odds of "Done" being printed twice go up dramatically:

```
static void Go()
{
  if (!_done) { Console.WriteLine ("Done"); _done = true; }
}
```

The problem is that one thread can be evaluating the `if` statement at exactly the same time as the other thread is executing the `WriteLine` statement—before it's had a chance to set `done` to `true`.

NOTE

Our example illustrates one of many ways that *shared writable state* can introduce the kind of intermittent errors for which multithreading is notorious. Next, we look at how to fix our program by locking; however, it's better to avoid shared state altogether where possible. We see later how asynchronous programming patterns help with this.

Locking and Thread Safety

NOTE

Locking and thread safety are large topics. For a full discussion, see "Exclusive Locking" and "Locking and Thread Safety" in Chapter 22.

We can fix the previous example by obtaining an *exclusive lock* while reading and writing to the shared field. C# provides the `lock` statement for just this purpose:

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    lock (_locker)
    {
      if (!_done) { Console.WriteLine ("Done"); _done = true; }
    }
  }
}
```

When two threads simultaneously contend a lock (which can be upon any reference-type object; in this case, `_locker`), one thread waits, or blocks, until the lock becomes available. In this case, it ensures that only one thread can enter its code block at a time, and “Done” will be printed just once. Code that’s protected in such a manner—from indeterminacy in a multithreaded context—is called *thread-safe*.

NOTE

Even the act of autoincrementing a variable is not thread-safe: the expression `x++` executes on the underlying processor as distinct read-increment-write operations. So, if two threads execute `x++` at once outside a lock, the variable can end up getting incremented once rather than twice (or worse, `x` could be *torn*, ending up with a bitwise mixture of old and new content, under certain conditions).

Locking is not a silver bullet for thread safety—it's easy to forget to lock around accessing a field, and locking can create problems of its own (such as deadlocking).

A good example of when you might use locking is around accessing a shared in-memory cache for frequently accessed database objects in an ASP.NET application. This kind of application is simple to get right, and there's no chance of deadlocking. We give an example in "Thread Safety in Application Servers".

Passing Data to a Thread

Sometimes, you'll want to pass arguments to the thread's startup method. The easiest way to do this is with a lambda expression that calls the method with the desired arguments:

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}
```

```
static void Print (string message) { Console.WriteLine (message); }
```

With this approach, you can pass in any number of arguments to the method. You can even wrap the entire implementation in a multistatement lambda:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

An alternative (and less flexible) technique is to pass an argument into `Thread`'s `Start` method:

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
  string message = (string) messageObj;   // We need to cast here
  Console.WriteLine (message);
}
```

This works because `Thread`'s constructor is overloaded to accept either of two delegates:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

LAMBDA EXPRESSIONS AND CAPTURED VARIABLES

As we saw, a lambda expression is the most convenient and powerful way to pass data to a thread. However, you must be careful about accidentally modifying *captured variables* after starting the thread. For instance, consider the following:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

The output is nondeterministic! Here’s a typical result:

```
0223557799
```

The problem is that the `i` variable refers to the *same* memory location throughout the loop’s lifetime. Therefore, each thread calls `Console.Write` on a variable whose value can change as it is running! The solution is to use a temporary variable as follows:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Each of the digits 0 to 9 is then written exactly once. (The *ordering* is still undefined because threads can start at indeterminate times.)

NOTE

This is analogous to the problem we described in “Captured Variables” in Chapter 8. The problem is just as much about C#’s rules for capturing variables in `for` loops as it is about multithreading.

Variable `temp` is now local to each loop iteration. Therefore, each thread captures a different memory location and there’s no problem. We can illustrate the problem in the earlier code more simply with the following example:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start(); t2.Start();
```

Because both lambda expressions capture the same text variable, `t2` is printed twice.

Exception Handling

Any `try`/`catch`/`finally` blocks in effect when a thread is created are of no relevance to the thread when it starts executing. Consider the following program:

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }
}

static void Go() { throw null; }   // Throws a NullReferenceException
```

The `try`/`catch` statement in this example is ineffective, and the newly created thread will be encumbered with an unhandled `NullReferenceException`. This behavior makes sense when you consider that each thread has an independent execution path.

The remedy is to move the exception handler into the `Go` method:

```
public static void Main()
{
   new Thread (Go).Start();
}

static void Go()
{
  try
  {
    ...
    throw null;    // The NullReferenceException will get caught below
    ...
  }
  catch (Exception ex)
  {
    // Typically log the exception, and/or signal another thread
    // that we've come unstuck
    ...
  }
}
```

You need an exception handler on all thread entry methods in production applications—just as you do (usually at a higher level, in the execution stack) on your main thread. An unhandled exception causes the whole application to shut down—with an ugly dialog box!

NOTE

In writing such exception handling blocks, rarely would you *ignore* the error: typically, you'd log the details of the exception. For a client application you might display a dialog box allowing the user to automatically submit those details to your web server. You then might choose to restart the application, because it's possible that an unexpected exception might leave your program in an invalid state.

CENTRALIZED EXCEPTION HANDLING

In WPF, UWP, and Windows Forms applications, you can subscribe to *global* exception handling events, `Application.DispatcherUnhandledException`, and `Application.ThreadException`, respectively. These fire after an unhandled exception in any part of your program that's called via the message loop (this amounts to all code that runs on the main thread while the `Application` is active). This is useful as a backstop for logging and reporting bugs (although it won't fire for unhandled exceptions on non-UI threads that you create). Handling these events prevents the program from shutting down, although you may choose to restart the application to avoid the potential corruption of state that can follow from (or that led to) the unhandled exception.

Foreground versus Background Threads

By default, threads you create explicitly are *foreground threads*. Foreground threads keep the application alive for as long as any one of them is running, whereas *background threads* do not. After all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

> **NOTE**
>
> A thread's foreground/background status has no relation to its *priority* (allocation of execution time).

You can query or change a thread's background status using its `IsBackground` property:

```
static void Main (string[] args)
{
  Thread worker = new Thread ( () => Console.ReadLine() );
  if (args.Length > 0) worker.IsBackground = true;
  worker.Start();
}
```

If this program is called with no arguments, the worker thread assumes foreground status and will wait on the `ReadLine` statement for the user to press Enter. Meanwhile, the main thread exits, but the application keeps running because a foreground thread is still alive.

On the other hand, if an argument is passed to `Main()`, the worker is assigned background status, and the program exits almost immediately as the main thread ends (terminating the `ReadLine`).

When a process terminates in this manner, any `finally` blocks in the execution stack of background threads are circumvented. If your program employs `finally` (or `using`) blocks to perform cleanup work such as deleting temporary files, you can avoid this by explicitly waiting out such background threads upon exiting an application, either by joining the thread, or with a signaling construct (see “Signaling”). In either case, you should specify a timeout, so you can abandon a renegade thread should it refuse to finish, otherwise your application will fail to close without the user having to enlist help from the Task Manager (or on Unix, the `kill` command).

Foreground threads don’t require this treatment, but you must take care to avoid bugs that could cause the thread not to end. A common cause for applications failing to exit properly is the presence of active foreground threads.

Thread Priority

A thread’s `Priority` property determines how much execution time it is allotted relative to other active threads in the OS, on the following scale:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest
}
```

This becomes relevant when multiple threads are simultaneously active. You need to take care when elevating a thread's priority because it can starve other threads. If you want a thread to have higher priority than threads in *other* processes, you must also elevate the process priority using the `Process` class in `System.Diagnostics`:

```
using Process p = Process.GetCurrentProcess();
p.PriorityClass = ProcessPriorityClass.High;
```

This can work well for non-UI processes that do minimal work and need low latency (the ability to respond very quickly) in the work they do. With compute-hungry applications (particularly those with a user interface), elevating process priority can starve other processes, slowing down the entire computer.

Signaling

Sometimes, you need a thread to wait until receiving notification(s) from other thread(s). This is called *signaling*. The simplest signaling construct is `ManualResetEvent`. Calling `WaitOne` on a `ManualResetEvent` blocks the current thread until another thread "opens" the signal by calling `Set`. In the following example, we start up a thread that waits on a `ManualResetEvent`. It remains blocked for two seconds until the main thread *signals* it:

```
var signal = new ManualResetEvent (false);

new Thread (() =>
{
  Console.WriteLine ("Waiting for signal...");
  signal.WaitOne();
  signal.Dispose();
  Console.WriteLine ("Got signal!");
}).Start();

Thread.Sleep(2000);
signal.Set();        // "Open" the signal
```

After calling `Set`, the signal remains open; you can close it again by calling `Reset`.

`ManualResetEvent` is one of several signaling constructs provided by the CLR; we cover all of them in detail in Chapter 22.

Threading in Rich Client Applications

In WPF, UWP, and Windows Forms applications, executing long-running operations on the main thread makes the application unresponsive because the main thread also processes the message loop that performs rendering and handles keyboard and mouse events.

A popular approach is to start up "worker" threads for time-consuming operations. The code on a worker thread runs a time-consuming operation and then updates the UI when complete. However, all rich client applications have a threading model whereby UI elements and controls can be accessed only from the thread that created them (typically the main UI thread). Violating this causes either unpredictable behavior, or an exception to be thrown.

Hence when you want to update the UI from a worker thread, you must forward the request to the UI thread (the technical term is *marshal*). The low-level way to do this is as follows (later, we discuss other solutions which build on these):

- In WPF, call `BeginInvoke` or `Invoke` on the element's `Dispatcher` object.
- In UWP apps, call `RunAsync` or `Invoke` on the `Dispatcher` object.
- In Windows Forms, call `BeginInvoke` or `Invoke` on the control.

All of these methods accept a delegate referencing the method you want to run. `BeginInvoke/RunAsync` work by enqueuing the delegate to the UI thread's *message queue* (the same queue that handles keyboard, mouse, and timer events). `Invoke` does the same thing, but then blocks until the message has been read and processed by the UI thread. Because of this, `Invoke` lets you get a return value back from the method. If you don't need a return value, `BeginInvoke`/`RunAsync` are preferable in that they don't block the caller and don't introduce the possibility of deadlock (see "Deadlocks" in Chapter 22).

NOTE

You can imagine that when you call `Application.Run`, the following pseudo-code executes:

```
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
    Keyboard/mouse message -> fire an event handler
    User BeginInvoke message -> execute delegate
    User Invoke message -> execute delegate & post
result
}
```

It's this kind of loop that enables a worker thread to marshal a delegate for execution onto the UI thread.

To demonstrate, suppose that we have a WPF window that contains a text box called `txtMessage`, whose content we want a worker thread to update after performing a time-consuming task (which we will simulate by calling `Thread.Sleep`). Here's how we'd do it:

```
partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000);           // Simulate time-consuming task
    UpdateMessage ("The answer");
  }

  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    Dispatcher.BeginInvoke (action);
  }
}
```

Running this results in a responsive window appearing immediately. Five seconds later, it updates the textbox. The code is similar for Windows Forms, except that we call the (`Form`’s) `BeginInvoke` method, instead:

```
  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    this.BeginInvoke (action);
  }
```

MULTIPLE UI THREADS

It’s possible to have multiple UI threads if they each own different windows. The main scenario is when you have an application with multiple top-level windows, often called a *Single Document Interface* (SDI) application, such as Microsoft Word. Each SDI window typically shows itself as a separate “application” on the taskbar and is mostly isolated, functionally, from other SDI windows. By giving each such window its own UI thread, each window can be made more responsive with respect to the others.

Synchronization Contexts

In the `System.ComponentModel` namespace, there's a class called `SynchronizationContext`, which enables the generalization of thread marshaling.

The rich-client APIs for mobile and desktop (UWP, WPF, and Windows Forms) each define and instantiate `SynchronizationContext` subclasses, which you can obtain via the static property `SynchronizationContext.Current` (while running on a UI thread). Capturing this property let you later *post* to UI controls from a worker thread:

```
partial class MyWindow : Window
{
  SynchronizationContext _uiSyncContext;

  public MyWindow()
  {
    InitializeComponent();
    // Capture the synchronization context for the current UI thread:
    _uiSyncContext = SynchronizationContext.Current;
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000);           // Simulate time-consuming task
    UpdateMessage ("The answer");
  }

  void UpdateMessage (string message)
  {
    // Marshal the delegate to the UI thread:
```

```
    _uiSyncContext.Post (_ => txtMessage.Text = message,
null);
  }
}
```

This is useful because the same technique works with all rich-client User Interface APIs.

Calling `Post` is equivalent to calling `BeginInvoke` on a `Dispatcher` or `Control`; there's also a `Send` method which is equivalent to `Invoke`.

The Thread Pool

Whenever you start a thread, a few hundred microseconds are spent organizing such things as a fresh local variable stack. The *thread pool* cuts this overhead by having a pool of pre-created recyclable threads. Thread pooling is essential for efficient parallel programming and fine-grained concurrency; it allows short operations to run without being overwhelmed with the overhead of thread startup.

There are a few things to be wary of when using pooled threads:

- You cannot set the `Name` of a pooled thread, making debugging more difficult (although you can attach a description when debugging in Visual Studio's Threads window).
- Pooled threads are always *background threads*.
- Blocking pooled threads can degrade performance (see "Hygiene in the thread pool").

You are free to change the priority of a pooled thread—it will be restored to normal when released back to the pool.

You can determine whether you’re currently executing on a pooled thread via the property `Thread.CurrentThread.IsThreadPoolThread`.

ENTERING THE THREAD POOL

The easiest way to explicitly run something on a pooled thread is to use `Task.Run` (we cover this in more detail in the following section):

```
// Task is in System.Threading.Tasks
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));
```

Because tasks didn’t exist prior to .NET Framework 4.0, a common alternative is to call `ThreadPool.QueueUserWorkItem`:

```
ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```

NOTE

The following use the thread pool implicitly:

- ASP.NET Core and Web API application servers
- `System.Timers.Timer` and `System.Threading.Timer`
- The parallel programming constructs that we describe in Chapter 23
- The (legacy) `BackgroundWorker` class

HYGIENE IN THE THREAD POOL

The thread pool serves another function, which is to ensure that a temporary excess of compute-bound work does not cause CPU *oversubscription*. Oversubscription is the condition of there being more active threads than CPU cores, with the OS having to time-slice threads. Oversubscription hurts performance because time-slicing requires expensive context switches and can invalidate the CPU caches that have become essential in delivering performance to modern processors.

The CLR avoids oversubscription in the thread pool by queuing tasks and throttling their startup. It begins by running as many concurrent tasks as there are hardware cores, and then tunes the level of concurrency via a hill-climbing algorithm, continually adjusting the workload in a particular direction. If throughput improves, it continues in the same direction (otherwise it reverses). This ensures that it always tracks the optimal performance curve—even in the face of competing process activity on the computer.

The CLR's strategy works best if two conditions are met:

- Work items are mostly short-running (<250 ms, or ideally <100 ms), so that the CLR has plenty of opportunities to measure and adjust.
- Jobs that spend most of their time blocked do not dominate the pool.

Blocking is troublesome because it gives the CLR the false idea that it's loading up the CPU. The CLR is smart enough to detect and compensate (by injecting more threads into the pool), although this can make the pool vulnerable to subsequent oversubscription. It also can introduce latency because the CLR throttles the rate at which it injects new threads, particularly early in an application's life (more so on client operating systems where it favors lower resource consumption).

Maintaining good hygiene in the thread pool is particularly relevant when you want to fully utilize the CPU (e.g., via the parallel programming APIs in Chapter 23).

Tasks

A thread is a low-level tool for creating concurrency, and as such, it has limitations. In particular:

- Although it's easy to pass data into a thread that you start, there's no easy way to get a "return value" back from a thread that you `Join`. You need to set up some kind of shared field. And if the operation throws an exception, catching and propagating that exception is equally painful.
- You can't tell a thread to start something else when it's finished; instead you must `Join` it (blocking your own thread in the process).

These limitations discourage fine-grained concurrency; in other words, they make it difficult to compose larger concurrent operations by combining smaller ones (something essential for the asynchronous programming that we look at in following sections). This in turn leads to greater reliance on manual synchronization (locking, signaling, and so on) and the problems that go with it.

The direct use of threads also has performance implications that we discussed in “The Thread Pool”. And should you need to run hundreds or thousands of concurrent I/O-bound operations, a thread-based approach consumes hundreds or thousands of megabytes of memory purely in thread overhead.

The `Task` class helps with all of these problems. Compared to a thread, a `Task` is higher-level abstraction—it represents a concurrent operation that might or might not be backed by a thread. Tasks are *compositional* (you can chain them together through the use of *continuations*). They can use the *thread pool* to lessen startup latency, and with a `TaskCompletionSource`, they can employ a callback approach that avoids threads altogether while waiting on I/O-bound operations.

The `Task` types were introduced in Framework 4.0 as part of the parallel programming library. However, they have since been enhanced (through the use of *awaiters*) to play equally well in more general concurrency scenarios and are backing types for C#’s asynchronous functions.

NOTE

In this section, we ignore the features of tasks that are aimed specifically at parallel programming; we cover them instead in Chapter 23.

Starting a Task

The easiest way to start a `Task` backed by a thread is with the static method `Task.Run` (the `Task` class is in the `System.Threading.Tasks` namespace). Simply pass in an `Action` delegate:

```
Task.Run (() => Console.WriteLine ("Foo"));
```

> **NOTE**
>
> Tasks use pooled threads by default, which are background threads. This means that when the main thread ends, so do any tasks that you create. Hence, to run these examples from a Console application, you must block the main thread after starting the task (for instance, by `Wait`ing the task or by calling `Console.ReadLine`):
>
> ```
> static void Main()
> {
>   Task.Run (() => Console.WriteLine ("Foo"));
>   Console.ReadLine();
> }
> ```
>
> In the book’s LINQPad companion samples, `Console.ReadLine` is omitted because the LINQPad process keeps background threads alive.

Calling `Task.Run` in this manner is similar to starting a thread as follows (except for the thread pooling implications that we discuss shortly):

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

`Task.Run` returns a `Task` object that we can use to monitor its progress, rather like a `Thread` object. (Notice, however, that we didn't call `Start` after calling `Task.Run` because this method creates "hot" tasks; you can instead use `Task`'s constructor to create "cold" tasks although this is rarely done in practice.)

You can track a task's execution status via its `Status` property.

WAIT

Calling `Wait` on a task blocks until it completes and is the equivalent of calling `Join` on a thread:

```
Task task = Task.Run (() =>
{
  Thread.Sleep (2000);
  Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted);  // False
task.Wait();  // Blocks until task is complete
```

`Wait` lets you optionally specify a timeout and a cancellation token to end the wait early (see "Cancellation").

LONG-RUNNING TASKS

By default, the CLR runs tasks on pooled threads, which is ideal for short-running compute-bound work. For longer-running and blocking operations (such as our preceding example), you can prevent use of a pooled thread as follows:

```
Task task = Task.Factory.StartNew (() => ...,

TaskCreationOptions.LongRunning);
```

NOTE

Running *one* long-running task on a pooled thread won't cause trouble; it's when you run multiple long-running tasks in parallel (particularly ones that block) that performance can suffer. And in that case, there are usually better solutions than `TaskCreationOptions.LongRunning`:

- If the tasks are I/O bound, `TaskCompletionSource` and *asynchronous functions* let you implement concurrency with callbacks (continuations) instead of threads.
- If the tasks are compute bound, a *producer/consumer queue* lets you throttle the concurrency for those tasks, avoiding starvation for other threads and processes (see "Writing a Producer/Consumer Queue" in Chapter 23).

Returning values

`Task` has a generic subclass called `Task<TResult>`, which allows a task to emit a return value. You can obtain a `Task<TResult>` by calling `Task.Run` with a `Func<TResult>` delegate (or a compatible lambda expression) instead of an `Action`:

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return
3; });
// ...
```

You can obtain the result later by querying the `Result` property. If the task hasn't yet finished, accessing this property will block the current thread until the task finishes:

```
int result = task.Result;      // Blocks if not already finished
Console.WriteLine (result);    // 3
```

In the following example, we create a task that uses LINQ to count the number of prime numbers in the first three million (+2) integers:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```

This writes "Task running...", and then a few seconds later, writes the answer of 216816.

NOTE

`Task<TResult>` can be thought of as a "future," in that it encapsulates a `Result` that becomes available later in time.

Exceptions

Unlike with threads, tasks conveniently propagate exceptions. So, if the code in your task throws an unhandled exception (in other words, if your task *faults*), that exception is automatically rethrown to whoever calls `Wait()`—or accesses the `Result` property of a `Task<TResult>`:

```
// Start a Task that throws a NullReferenceException:
Task task = Task.Run (() => { throw null; });
try
{
  task.Wait();
}
catch (AggregateException aex)
{
  if (aex.InnerException is NullReferenceException)
    Console.WriteLine ("Null!");
  else
    throw;
}
```

(The CLR wraps the exception in an `AggregateException` in order to play well with parallel programming scenarios; we discuss this in Chapter 23.)

You can test for a faulted task without rethrowing the exception via the `IsFaulted` and `IsCanceled` properties of the `Task`. If both properties return `false`, no error occurred; if `IsCanceled` is `true`, an `OperationCanceledException` was thrown for that task (see "Cancellation"); if `IsFaulted` is `true`, another type of exception was thrown and the `Exception` property will indicate the error.

EXCEPTIONS AND AUTONOMOUS TASKS

With autonomous "set-and-forget" tasks (those for which you don't rendezvous via `Wait()` or `Result`, or a continuation that does the same), it’s good practice to explicitly exception-handle the task code to avoid silent failure, just as you would with a thread.

NOTE

Ignoring exceptions is fine when an exception solely indicates a failure to obtain a result that you’re no longer interested in. For example, if a user cancels a request to download a web page, we wouldn’t care if it turns out that the web page didn’t exist.

Ignoring exceptions is problematic when an exception indicates a bug in your program, for two reasons:

- The bug may have left your program in an invalid state.
- More exceptions may occur later as a result of the bug, and failure to log the initial error can make diagnosis difficult.

You can subscribe to unobserved exceptions at a global level via the static event `TaskScheduler.UnobservedTaskException`; handling this event and logging the error can make good sense.

There are a couple of interesting nuances on what counts as unobserved:

- Tasks waited upon with a timeout will generate an unobserved exception if the faults occurs *after* the timeout interval.
- The act of checking a task’s `Exception` property after it has faulted makes the exception “observed.”

Continuations

A continuation says to a task, “when you’ve finished, continue by doing something else.” A continuation is usually implemented by a callback that executes once upon completion of an operation. There are two ways to attach a continuation to a task. The first is particularly significant because it’s used by C#’s asynchronous functions, as you’ll see soon. We can demonstrate it with the prime number counting task that we wrote a short while ago in “Returning values”:

```
Task<int> primeNumberTask = Task.Run (() =>
  Enumerable.Range (2, 3000000).Count (n =>
    Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted (() =>
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);       // Writes result
});
```

Calling `GetAwaiter` on the task returns an *awaiter* object whose `OnCompleted` method tells the *antecedent* task (`primeNumberTask`) to execute a delegate when it finishes (or faults). It’s valid to attach a continuation to an already-completed task, in which case the continuation will be scheduled to execute right away.

NOTE

An *awaiter* is any object that exposes the two methods that we've just seen (`OnCompleted` and `GetResult`), and a Boolean property called `IsCompleted`. There's no interface or base class to unify all of these members (although `OnCompleted` is part of the interface `INotifyCompletion`). We explain the significance of the pattern in "Asynchronous Functions in C#".

If an antecedent task faults, the exception is rethrown when the continuation code calls `awaiter.GetResult()`. Rather than calling `GetResult`, we could simply access the `Result` property of the antecedent. The benefit of calling `GetResult` is that if the antecedent faults, the exception is thrown directly without being wrapped in `AggregateException`, allowing for simpler and cleaner `catch` blocks.

For nongeneric tasks, `GetResult()` has a void return value. Its useful function is then solely to rethrow exceptions.

If a synchronization context is present, `OnCompleted` automatically captures it and posts the continuation to that context. This is very useful in rich client applications because it bounces the continuation back to the UI thread. In writing libraries, however, it's not usually desirable because the relatively expensive UI-thread-bounce should occur just once upon leaving the library rather than between method calls. Hence, you can defeat it by using the `ConfigureAwait` method:

```
var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();
```

If no synchronization context is present—or you use `ConfigureAwait(false)`—the continuation will (in general) execute on the same thread as the antecedent, avoiding unnecessary overhead.

The other way to attach a continuation is by calling the task's `ContinueWith` method:

```
primeNumberTask.ContinueWith (antecedent =>
{
  int result = antecedent.Result;
  Console.WriteLine (result);          // Writes 123
});
```

`ContinueWith` itself returns a `Task`, which is useful if you want to attach further continuations. However, you must deal directly with `AggregateException` if the task faults, and write extra code to marshal the continuation in UI applications (see "Task Schedulers" in Chapter 23). And in non-UI contexts, you must specify `TaskContinuationOptions.ExecuteSynchronously` if you want the continuation to execute on the same thread; otherwise it will bounce to the thread pool. `ContinueWith` is particularly useful in parallel programming scenarios; we cover it in detail in "Continuations" in Chapter 23.

TaskCompletionSource

We've seen how `Task.Run` creates a task that runs a delegate on a pooled (or non-pooled) thread. Another way to create a task is with `TaskCompletionSource`.

`TaskCompletionSource` lets you create a task out of any operation that starts and finishes some time later. It works by giving you a “slave” task that you manually drive—by indicating when the operation finishes or faults. This is ideal for I/O-bound work: you get all the benefits of tasks (with their ability to propagate return values, exceptions, and continuations) without blocking a thread for the duration of the operation.

To use `TaskCompletionSource`, you simply instantiate the class. It exposes a `Task` property that returns a task upon which you can wait and attach continuations—just as with any other task. The task, however, is controlled entirely by the `TaskCompletionSource` object via the following methods:

```
public class TaskCompletionSource<TResult>
{
  public void SetResult (TResult result);
  public void SetException (Exception exception);
  public void SetCanceled();

  public bool TrySetResult (TResult result);
  public bool TrySetException (Exception exception);
  public bool TrySetCanceled();
  public bool TrySetCanceled (CancellationToken cancellationToken);
  ...
}
```

Calling any of these methods *signals* the task, putting it into a completed, faulted, or canceled state (we cover the latter in the section “Cancellation”). You’re supposed to call one of these methods exactly once: if called again, `SetResult`, `SetException`, or

`SetCanceled` will throw an exception, whereas the `Try*` methods return `false`.

The following example prints 42 after waiting for five seconds:

```
var tcs = new TaskCompletionSource<int>();

new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); })
  { IsBackground = true }
  .Start();

Task<int> task = tcs.Task;         // Our "slave" task.
Console.WriteLine (task.Result);   // 42
```

With `TaskCompletionSource`, we can write our own `Run` method:

```
Task<TResult> Run<TResult> (Func<TResult> function)
{
  var tcs = new TaskCompletionSource<TResult>();
  new Thread (() =>
  {
    try { tcs.SetResult (function()); }
    catch (Exception ex) { tcs.SetException (ex); }
  }).Start();
  return tcs.Task;
}
...
Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });
```

Calling this method is equivalent to calling `Task.Factory.StartNew` with the `TaskCreationOptions.LongRunning` option to request a nonpooled thread.

The real power of `TaskCompletionSource` is in creating tasks that don't tie up threads. For instance, consider a task that waits for five seconds and then returns the number 42. We can write this without a thread by using the `Timer` class, which with the help of the CLR (and in turn, the OS) fires an event in *x* milliseconds (we revisit timers in Chapter 22):

```
Task<int> GetAnswerToLife()
{
  var tcs = new TaskCompletionSource<int>();
  // Create a timer that fires once in 5000 ms:
  var timer = new System.Timers.Timer (5000) { AutoReset =
false };
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult
(42); };
  timer.Start();
  return tcs.Task;
}
```

Hence, our method returns a task that completes five seconds later, with a result of 42. By attaching a continuation to the task, we can write its result without blocking *any* thread:

```
var awaiter = GetAnswerToLife().GetAwaiter();
awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult()));
```

We could make this more useful and turn it into a general-purpose `Delay` method by parameterizing the delay time and getting rid of the return value. This means having it return a `Task` instead of a `Task<int>`. However, there's no nongeneric version of `TaskCompletionSource`, which means we can't directly create a nongeneric `Task`. The workaround is simple: because `Task<TResult>` derives from `Task`, we create a `TaskCompletionSource<`*`anything`*`>` and then implicitly convert the `Task<`*`anything`*`>` that it gives us into a `Task`, like this:

```
var tcs = new TaskCompletionSource<object>();
Task task = tcs.Task;
```

Now we can write our general-purpose `Delay` method:

```
Task Delay (int milliseconds)
{
  var tcs = new TaskCompletionSource<object>();
  var timer = new System.Timers.Timer (milliseconds) { AutoReset = false
};
  timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult (null);
};
  timer.Start();
  return tcs.Task;
}
```

Here’s how we can use it to write “42” after five seconds:

```
Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

Our use of `TaskCompletionSource` without a thread means that a thread is engaged only when the continuation starts, five seconds later. We can demonstrate this by starting 10,000 of these operations at once without error or excessive resource consumption:

```
for (int i = 0; i < 10000; i++)
  Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine (42));
```

NOTE

Timers fire their callbacks on pooled threads, so after five seconds, the thread pool will receive 10,000 requests to call `SetResult(null)` on a `TaskCompletionSource`. If the requests arrive faster than they can be processed, the thread pool will respond by enqueuing and then processing them at the optimum level of parallelism for the CPU. This is ideal if the thread-bound jobs are short running, which is true in this case: the thread-bound job is merely the call to `SetResult` plus either the action of posting the continuation to the synchronization context (in a UI application) or otherwise the continuation itself (`Console.WriteLine(42)`).

Task.Delay

The `Delay` method that we just wrote is sufficiently useful that it's available as a static method on the `Task` class

```
Task.Delay (5000).GetAwaiter().OnCompleted (() => Console.WriteLine
(42));
```

or:

```
Task.Delay (5000).ContinueWith (ant => Console.WriteLine (42));
```

`Task.Delay` is the *asynchronous* equivalent of `Thread.Sleep`.

Principles of Asynchrony

In demonstrating `TaskCompletionSource`, we ended up writing *asynchronous* methods. In this section, we define exactly what asynchronous operations are and explain how this leads to asynchronous programming.

Synchronous versus Asynchronous Operations

A *synchronous operation* does its work *before* returning to the caller.

An *asynchronous operation* can do (most or all of) its work *after* returning to the caller.

The majority of methods that you write and call are synchronous. An example is `List<T>.Add`, or `Console.WriteLine`, or `Thread.Sleep`. Asynchronous methods are less common and initiate *concurrency*, because work continues in parallel to the caller. Asynchronous methods typically return quickly (or immediately) to the caller; thus, they are also called *nonblocking methods*.

Most of the asynchronous methods that we've seen so far can be described as general-purpose methods:

- `Thread.Start`
- `Task.Run`
- Methods that attach continuations to tasks

In addition, some of the methods that we discussed in "Synchronization Contexts" (`Dispatcher.BeginInvoke`, `Control.BeginInvoke`, and `SynchronizationContext.Post`) are asynchronous, as are the methods that we wrote in "TaskCompletionSource", including `Delay`.

What Is Asynchronous Programming?

The principle of asynchronous programming is that you write long-running (or potentially long-running) functions asynchronously. This is in contrast to the conventional approach of writing long-running functions synchronously, and then calling those functions from a new thread or task to introduce concurrency as required.

The difference with the asynchronous approach is that concurrency is initiated *inside* the long-running function rather than from *outside* the function. This has two benefits:

- I/O-bound concurrency can be implemented without tying up threads (as we demonstrate in "TaskCompletionSource"), improving scalability and efficiency.
- Rich-client applications end up with less code on worker threads, simplifying thread safety.

This, in turn, leads to two distinct uses for asynchronous programming. The first is writing (typically server-side) applications that deal efficiently with a lot of concurrent I/O. The challenge here is not thread *safety* (because there's usually minimal shared state) but thread *efficiency*; in particular, not consuming a thread per network request. So, in this context, it's only I/O-bound operations that benefit from asynchrony.

The second use is to simplify thread safety in rich-client applications. This is particularly relevant as a program grows in size, because to deal with complexity, we typically refactor larger methods into smaller ones, resulting in chains of methods that call one another (*call graphs*).

With a traditional *synchronous* call graph, if any operation within the graph is long-running, we must run the entire call graph on a worker thread to maintain a responsive UI. Hence, we end up with a single concurrent operation that spans many methods (*coarse-grained concurrency*), and this requires considering thread safety for every method in the graph.

With an *asynchronous* call graph, we need not start a thread until it's actually needed, typically low in the graph (or not at all in the case of I/O-bound operations). All other methods can run entirely on the UI thread, with much-simplified thread safety. This results in *fine-grained concurrency*—a sequence of small concurrent operations, between which execution bounces to the UI thread.

NOTE

To benefit from this, both I/O- and compute-bound operations need to be written asynchronously; a good rule of thumb is to include anything that might take longer than 50 ms.

(On the flip side, *excessively* fine-grained asynchrony can hurt performance, because asynchronous operations incur an overhead—see "Optimizations".)

In this chapter, we focus mostly on the rich-client scenario, which is the more complex of the two. In Chapter 16, we give two examples that illustrate the I/O-bound scenario (see “Concurrency with TCP” and “Writing an HTTP Server”).

> **NOTE**
>
> The UWP framework encourages asynchronous programming to the point where synchronous versions of some long-running methods are either not exposed or throw exceptions. Instead, you must call asynchronous methods that return tasks (or objects that can be converted into tasks via the `AsTask` extension method).

Asynchronous Programming and Continuations

Tasks are ideally suited to asynchronous programming, because they support continuations, which are essential for asynchrony (consider the `Delay` method that we wrote in “TaskCompletionSource”). In writing `Delay`, we used `TaskCompletionSource`, which is a standard way to implement “bottom-level” I/O-bound asynchronous methods.

For compute-bound methods, we use `Task.Run` to initiate thread-bound concurrency. Simply by returning the task to the caller, we create an asynchronous method. What distinguishes asynchronous programming is that we aim to do so lower in the call graph, so that in rich-client applications, higher-level methods can remain on the UI thread and access controls and shared state without thread-safety issues. To illustrate, consider the following method that computes and counts prime numbers, using all available cores (we discuss `ParallelEnumerable` in Chapter 23):

```
int GetPrimesCount (int start, int count)
{
  return
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0));
}
```

The details of how this works are unimportant; what matters is that it can take a while to run. We can demonstrate this by writing another method to call it:

```
void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (GetPrimesCount (i*1000000 + 2, 1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

Here’s the output:

```
78498 primes between 0 and 999999
70435 primes between 1000000 and 1999999
67883 primes between 2000000 and 2999999
66330 primes between 3000000 and 3999999
65367 primes between 4000000 and 4999999
64336 primes between 5000000 and 5999999
63799 primes between 6000000 and 6999999
63129 primes between 7000000 and 7999999
```

```
62712 primes between 8000000 and 8999999
62090 primes between 9000000 and 9999999
```

Now we have a *call graph*, with `DisplayPrimeCounts` calling `GetPrimesCount`. The former uses `Console.WriteLine` for simplicity, although in reality it would more likely be updating UI controls in a rich-client application, as we demonstrate later. We can initiate coarse-grained concurrency for this call graph as follows:

```
Task.Run (() => DisplayPrimeCounts());
```

With a fine-grained asynchronous approach, we instead start by writing an asynchronous version of `GetPrimesCount`:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

Why Language Support Is Important

Now we must modify `DisplayPrimeCounts` so that it calls `GetPrimesCount`**`Async`**. This is where C#'s `await` and `async` keywords come into play, because to do so otherwise is trickier than it sounds. If we simply modify the loop as follows:

```
for (int i = 0; i < 10; i++)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2,
1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
    Console.WriteLine (awaiter.GetResult() + " primes between... "));
}
Console.WriteLine ("Done");
```

the loop will rapidly spin through 10 iterations (the methods being nonblocking) and all 10 operations will execute in parallel (followed by a premature “Done”).

NOTE

Executing these tasks in parallel is undesirable in this case because their internal implementations are already parallelized; it will only make us wait longer to see the first results (and muck up the ordering).

There is a much more common reason, however, for needing to *serialize* the execution of tasks, which is that Task B depends on the result of Task A. For example, in fetching a web page, a DNS lookup must precede the HTTP request.

To get them running sequentially, we must trigger the next loop iteration from the continuation itself. This means eliminating the `for` loop and resorting to a recursive call in the continuation:

```
void DisplayPrimeCounts()
{
  DisplayPrimeCountsFrom (0);
}

void DisplayPrimeCountsFrom (int i)
{
  var awaiter = GetPrimesCountAsync (i*1000000 + 2,
1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    Console.WriteLine (awaiter.GetResult() + " primes between...");
    if (++i < 10) DisplayPrimeCountsFrom (i);
    else Console.WriteLine ("Done");
  });
}
```

It gets even worse if we want to make `DisplayPrimesCount` *itself* asynchronous, returning a task that it signals upon completion. To accomplish this requires creating a `TaskCompletionSource`:

```
Task DisplayPrimeCountsAsync()
{
  var machine = new PrimesStateMachine();
  machine.DisplayPrimeCountsFrom (0);
  return machine.Task;
}

class PrimesStateMachine
{
  TaskCompletionSource<object> _tcs = new
TaskCompletionSource<object>();
  public Task Task { get { return _tcs.Task; } }

  public void DisplayPrimeCountsFrom (int i)
  {
    var awaiter = GetPrimesCountAsync (i*1000000+2,
1000000).GetAwaiter();
    awaiter.OnCompleted (() =>
    {
      Console.WriteLine (awaiter.GetResult());
      if (++i < 10) DisplayPrimeCountsFrom (i);
      else { Console.WriteLine ("Done"); _tcs.SetResult (null); }
    });
  }
}
```

Fortunately, C#’s *asynchronous functions* do all of this work for us. With the `async` and `await` keywords, we need only write this:

```
async Task DisplayPrimeCountsAsync()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000 + 2,
1000000) +
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1));
  Console.WriteLine ("Done!");
}
```

Consequently, `async` and `await` are essential for implementing asynchrony without excessive complexity. Let’s now see how these keywords work.

NOTE

Another way of looking at the problem is that imperative looping constructs (`for`, `foreach`, and so on) do not mix well with continuations, because they rely on the *current local state* of the method (“how many more times is this loop going to run?”).

Although the `async` and `await` keywords offer one solution, it’s sometimes possible to solve it in another way by replacing the imperative looping constructs with the *functional* equivalent (in other words, LINQ queries). This is the basis of *Reactive Framework* (Rx) and can be a good option when you want to execute query operators over the result—or combine multiple sequences. The price to pay is that to avoid blocking, Rx operates over *push*-based sequences, which can be conceptually tricky.

Asynchronous Functions in C#

The `async` and `await` keywords let you write asynchronous code that has the same structure and simplicity as synchronous code while eliminating the “plumbing” of asynchronous programming.

Awaiting

The `await` keyword simplifies the attaching of continuations. Starting with a basic scenario, the compiler expands this:

```
var result = await expression;
statement(s);
```

into something functionally similar to this:

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var result = awaiter.GetResult();
  statement(s);
});
```

NOTE

The compiler also emits code to short-circuit the continuation in case of synchronous completion (see “Optimizations”) and to handle various nuances that we pick up in later sections.

To demonstrate, let’s revisit the asynchronous method that we wrote previously that computes and counts prime numbers:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));
}
```

With the `await` keyword, we can call it as follows:

```
int result = await GetPrimesCountAsync (2, 1000000);
Console.WriteLine (result);
```

To compile, we need to add the `async` modifier to the containing method:

```
async void DisplayPrimesCount()
{
  int result = await GetPrimesCountAsync (2, 1000000);
  Console.WriteLine (result);
}
```

The `async` modifier instructs the compiler to treat `await` as a keyword rather than an identifier should an ambiguity arise within that method (this ensures that code written prior to C# 5 that might use `await` as an identifier will still compile without error). The `async` modifier can be applied only to methods (and lambda expressions) that return `void` or (as you’ll see later) a `Task` or `Task<TResult>`.

NOTE

The `async` modifier is similar to the `unsafe` modifier in that it has no effect on a method’s signature or public metadata; it affects only what happens *inside* the method. For this reason, it makes no sense to use `async` in an interface. However, it is legal, for instance, to introduce `async` when overriding a non-`async` virtual method, as long as you keep the signature the same.

Methods with the `async` modifier are called *asynchronous functions*, because they themselves are typically asynchronous. To see why, let’s look at how execution proceeds through an asynchronous function.

Upon encountering an `await` expression, execution (normally) returns to the caller—rather like with `yield return` in an iterator. But before returning, the runtime attaches a continuation to the awaited task, ensuring that when the task completes, execution jumps back into the method and continues where it left off. If the task faults, its exception is rethrown, otherwise its return value is assigned to the `await` expression. We can summarize everything we just said by looking at the logical expansion of the asynchronous method we just examined:

```
void DisplayPrimesCount()
{
  var awaiter = GetPrimesCountAsync (2, 1000000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    int result = awaiter.GetResult();
    Console.WriteLine (result);
  });
}
```

The expression upon which you `await` is typically a task; however, any object with a `GetAwaiter` method that returns an *awaiter* (implementing `INotifyCompletion.OnCompleted` and with an appropriately typed `GetResult` method and a `bool IsCompleted` property) will satisfy the compiler.

Notice that our `await` expression evaluates to an `int` type; this is because the expression that we awaited was a `Task<int>` (whose `GetAwaiter().GetResult()` method returns an `int`).

Awaiting a nongeneric task is legal and generates a void expression:

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

CAPTURING LOCAL STATE

The real power of `await` expressions is that they can appear almost anywhere in code. Specifically, an `await` expression can appear in place of any expression (within an asynchronous function) except for inside a `lock` expression or `unsafe` context.

In the following example, we `await` inside a loop:

```
async void DisplayPrimeCounts()
{
  for (int i = 0; i < 10; i++)
    Console.WriteLine (await GetPrimesCountAsync (i*1000000+2,
1000000));
}
```

Upon first executing `GetPrimesCountAsync`, execution returns to the caller by virtue of the `await` expression. When the method completes (or faults), execution resumes where it left off, with the values of local variables and loop counters preserved.

Without the `await` keyword, the simplest equivalent might be the example we wrote in "Why Language Support Is Important". The compiler, however, takes the more general strategy of refactoring such methods into state machines (rather like it does with iterators).

The compiler relies on continuations (via the awaiter pattern) to resume execution after an `await` expression. This means that if running on the UI thread of a rich-client application, the synchronization context ensures execution resumes on the same thread. Otherwise, execution resumes on whatever thread the task finished on. The change-of-thread does not affect the order of execution and is of little consequence unless you're somehow relying on thread affinity, perhaps through the use of thread-local storage (see "Thread-Local Storage" in Chapter 22). It's like touring a city and hailing taxis to get from one destination to another. With a synchronization context, you'll always get the same taxi; with no synchronization context, you'll usually get a different taxi each time. In either case, though, the journey is the same.

AWAITING IN A UI

We can demonstrate asynchronous functions in a more practical context by writing a simple UI that remains responsive while calling a compute-bound method. Let's begin with a synchronous solution:

```
class TestUI : Window
{
  Button _button = new Button { Content = "Go" };
  TextBlock _results = new TextBlock();

  public TestUI()
  {
    var panel = new StackPanel();
    panel.Children.Add (_button);
    panel.Children.Add (_results);
    Content = panel;
    _button.Click += (sender, args) => Go();
  }

  void Go()
  {
    for (int i = 1; i < 5; i++)
      _results.Text += GetPrimesCount (i * 1000000, 1000000) +
        " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
        Environment.NewLine;
  }

  int GetPrimesCount (int start, int count)
  {
    return ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0));
  }
}
```

Upon pressing the “Go” button, the application becomes unresponsive for the time it takes to execute the compute-bound code. There are two steps in asynchronizing this; the first is to switch to the asynchronous version of `GetPrimesCount` that we used in previous examples:

```
Task<int> GetPrimesCountAsync (int start, int count)
{
  return Task.Run (() =>
    ParallelEnumerable.Range (start, count).Count (n =>
      Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i >
0)));
}
```

The second step is to modify `Go` to call `GetPrimesCountAsync`:

```
async void Go()
{
  _button.IsEnabled = false;
  for (int i = 1; i < 5; i++)
    _results.Text += await GetPrimesCountAsync (i * 1000000, 1000000)
+
      " primes between " + (i*1000000) + " and " + ((i+1)*1000000-1) +
      Environment.NewLine;
  _button.IsEnabled = true;
}
```

This illustrates the simplicity of programming with asynchronous functions: you program as you would synchronously, but call asynchronous functions instead of blocking functions and `await` them. Only the code within `GetPrimesCountAsync` runs on a worker thread; the code in `Go` "leases" time on the UI thread. We could say that `Go` executes *pseudoconcurrently* to the message loop (in that its execution is interspersed with other events that the UI thread processes). With this pseudoconcurrency, the only point at which preemption can occur is during an `await`. This simplifies thread safety: in our case, the only problem that this could cause is *reentrancy* (clicking the button again while it's running, which we avoid by disabling the button). True concurrency occurs lower in the call stack, inside code called by `Task.Run`. To benefit from this model, truly concurrent code avoids accessing shared state or UI controls.

To give another example, suppose that instead of calculating prime numbers, we want to download several web pages and sum their lengths. .NET Core exposes numerous task-returning asynchronous methods, one of which is the `WebClient` class in `System.Net`. The `DownloadDataTaskAsync` method asynchronously downloads a URI to a byte array, returning a `Task<byte[]>`, so by awaiting it, we get a `byte[]`. Let’s now rewrite our `Go` method:

```
async void Go()
{
  _button.IsEnabled = false;
  string[] urls = "www.albahari.com www.oreilly.com
www.linqpad.net".Split();
  int totalLength = 0;
  try
  {
    foreach (string url in urls)
    {
      var uri = new Uri ("http://" + url);
      byte[] data = await new WebClient().DownloadDataTaskAsync
(uri);
      _results.Text += "Length of " + url + " is " + data.Length +
                        Environment.NewLine;
      totalLength += data.Length;
    }
    _results.Text += "Total length: " + totalLength;
  }
  catch (WebException ex)
  {
    _results.Text += "Error: " + ex.Message;
  }
  finally { _button.IsEnabled = true; }
}
```

Again, this mirrors how we'd write it synchronously—including the use of `catch` and `finally` blocks. Even though execution returns to the caller after the first `await`, the `finally` block does not execute until the method has logically completed (by virtue of all its code executing—or an early `return` or unhandled exception).

It can be helpful to consider exactly what's happening underneath. First, we need to revisit the pseudocode that runs the message loop on the UI thread:

```
Set synchronization context for this thread to WPF sync context
while (!thisApplication.Ended)
{
  wait for something to appear in message queue
  Got something: what kind of message is it?
    Keyboard/mouse message -> fire an event handler
    User BeginInvoke/Invoke message -> execute delegate
}
```

Event handlers that we attach to UI elements execute via this message loop. When our `Go` method runs, execution proceeds as far as the `await` expression, and then returns to the message loop (freeing the UI to respond to further events). However, the compiler's expansion of `await` ensures that before returning, a continuation is set up such that execution resumes where it left off upon completion of the task. And because we awaited on a UI thread, the continuation posts to the synchronization context which executes it via the message loop, keeping our entire `Go` method executing pseudo-concurrently on the UI thread. True (I/O-bound) concurrency occurs within the implementation of `DownloadDataTaskAsync`.

COMPARISON TO COARSE-GRAINED CONCURRENCY

Asynchronous programming was difficult prior to C# 5, not only because there was no language support, but also because the .NET Framework exposed asynchronous functionality through clumsy patterns called the EAP and the APM (see "Obsolete Patterns") rather than task-returning methods.

The popular workaround was coarse-grained concurrency (in fact, there was even a type called `BackgroundWorker` to help with that). Returning to our original *synchronous* example with `GetPrimesCount`, we can demonstrate coarse-grained asynchrony by modifying the button's event handler, as follows:

```
...
_button.Click += (sender, args) =>
{
  _button.IsEnabled = false;
  Task.Run (() => Go());
};
```

(We've chosen to use `Task.Run` rather than `BackgroundWorker` because the latter would do nothing to simplify our particular example.) In either case, the end result is that our entire synchronous call graph (`Go` plus `GetPrimesCount`) runs on a worker thread. And because `Go` updates UI elements, we must now litter our code with `Dispatcher.BeginInvoke`:

```
void Go()
{
  for (int i = 1; i < 5; i++)
  {
    int result = GetPrimesCount (i * 1000000, 1000000);
    Dispatcher.BeginInvoke (new Action (() =>
      _results.Text += result + " primes between " + (i*1000000) +
      " and " + ((i+1)*1000000-1) + Environment.NewLine));
  }
  Dispatcher.BeginInvoke (new Action (() => _button.IsEnabled =
true));
}
```

Unlike with the asynchronous version, the loop itself runs on a worker thread. This might seem innocuous, and yet, even in this simple case, our use of multithreading has introduced a race condition. (Can you spot it? If not, try running the program: it will almost certainly become apparent.)

Implementing cancellation and progress reporting creates more possibilities for thread-safety errors, as does any additional code in the method. For instance, suppose that the upper limit for the loop is not hardcoded, but comes from a method call:

```
  for (int i = 1; i < GetUpperBound(); i++)
```

Now suppose that `GetUpperBound()` reads the value from a lazily-loaded configuration file, which loads from disk upon first call. All of this code now runs on your worker thread, code that's most likely not thread-safe. This is the danger of starting worker threads high in the call graph.

Writing Asynchronous Functions

With any asynchronous function, you can replace the `void` return type with a `Task` to make the method itself *usefully* asynchronous (and `await`able). No further changes are required:

```
async Task PrintAnswerToLife()   // We can return Task instead of void
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Notice that we don't explicitly return a task in the method body. The compiler manufactures the task, which it signals upon completion of the method (or upon an unhandled exception). This makes it easy to create asynchronous call chains:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
```

And because we've declared `Go` with a `Task` return type, `Go` itself is awaitable.

The compiler expands asynchronous functions that return tasks into code that uses `TaskCompletionSource` to create a task that it then signals or faults.

Nuances aside, we can expand `PrintAnswerToLife` into the following functional equivalent:

```
Task PrintAnswerToLife()
{
  var tcs = new TaskCompletionSource<object>();
  var awaiter = Task.Delay (5000).GetAwaiter();
  awaiter.OnCompleted (() =>
  {
    try
    {
      awaiter.GetResult();    // Re-throw any exceptions
      int answer = 21 * 2;
      Console.WriteLine (answer);
      tcs.SetResult (null);
    }
    catch (Exception ex) { tcs.SetException (ex); }
  });
  return tcs.Task;
}
```

Hence, whenever a task-returning asynchronous method finishes, execution jumps back to whatever awaited it (by virtue of a continuation).

NOTE

In a rich-client scenario, execution bounces at this point back to the UI thread (if it's not already on the UI thread). Otherwise, it continues on whatever thread the continuation came back on. This means that there's no latency cost in bubbling up asynchronous call graphs, other than the first "bounce" if it was UI-thread-initiated.

RETURNING TASK<TRESULT>

You can return a `Task<TResult>` if the method body returns `TResult`:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;    // Method has return type Task<int> we return int
}
```

Internally, this results in the `TaskCompletionSource` being signaled with a value rather than null. We can demonstrate `GetAnswerToLife` by calling it from `PrintAnswerToLife` (which in turn, called from `Go`):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
```

```
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```

In effect, we’ve refactored our original `PrintAnswerToLife` into two methods—with the same ease as if we were programming synchronously. The similarity to synchronous programming is intentional; here’s the synchronous equivalent of our call graph, for which calling `Go()` gives the same result after blocking for five seconds:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Done");
}

void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}

int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

> **NOTE**
>
> This also illustrates the basic principle of how to design with asynchronous functions in C#:
>
> 1. Write your methods synchronously.
> 2. Replace *synchronous* method calls with *asynchronous* method calls, and `await` them.
> 3. Except for “top-level” methods (typically event handlers for UI controls), upgrade your asynchronous methods’ return types to `Task` or `Task<TResult>` so that they’re awaitable.

The compiler’s ability to manufacture tasks for asynchronous functions means that for the most part, you need to explicitly instantiate a `TaskCompletionSource` only in (the relatively rare case of) bottom-level methods that initiate I/O-bound concurrency. (And for methods that initiate compute-bound concurrency, you create the task with `Task.Run`.)

ASYNCHRONOUS CALL GRAPH EXECUTION

To see exactly how this executes, it’s helpful to rearrange our code as follows:

```
async Task Go()
{
  var task = PrintAnswerToLife();
  await task; Console.WriteLine ("Done");
}

async Task PrintAnswerToLife()
{
  var task = GetAnswerToLife();
  int answer = await task; Console.WriteLine (answer);
}

async Task<int> GetAnswerToLife()
{
  var task = Task.Delay (5000);
  await task; int answer = 21 * 2; return answer;
}
```

`Go` calls `PrintAnswerToLife`, which calls `GetAnswerToLife`, which calls `Delay` and then awaits. The `await` causes execution to return to `PrintAnswerToLife`, which itself awaits, returning to `Go`, which also awaits and returns to the caller. All of this happens synchronously on the thread that called `Go`; this is the brief *synchronous* phase of execution.

Five seconds later, the continuation on `Delay` fires and execution returns to `GetAnswerToLife` on a pooled thread. (If we started on a UI thread, execution now bounces to that thread.) The remaining statements in `GetAnswerToLife` then run, after which the method's `Task<int>` completes with a result of 42 and executes the continuation in `PrintAnswerToLife`, which executes the remaining statements in that method. The process continues until `Go`'s task is signaled as complete.

Execution flow matches the synchronous call graph that we showed earlier because we're following a pattern whereby we `await` every asynchronous method immediately after calling it. This creates a sequential flow with no parallelism or overlapping execution within the call graph. Each `await` expression creates a *gap* in execution, after which the program resumes where it left off.

PARALLELISM

Calling an asynchronous method without awaiting it allows the code that follows to execute in parallel. You might have noticed in earlier examples that we had a button whose event handler called `Go`, as follows:

```
_button.Click += (sender, args) => Go();
```

Despite `Go` being an asynchronous method, we didn't await it, and this is indeed what facilitates the concurrency needed to maintain a responsive UI.

We can use this same principle to run two asynchronous operations in parallel:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

(By awaiting both operations afterward, we "end" the parallelism at that point. Later, we describe how the `WhenAll` task combinator helps with this pattern.)

Concurrency created in this manner occurs whether or not the operations are initiated on a UI thread, although there’s a difference in how it occurs. In both cases, we get the same “true” concurrency occurring in the bottom-level operations that initiate it (such as `Task.Delay`, or code farmed to `Task.Run`). Methods above this in the call stack will be subject to true concurrency only if the operation was initiated without a synchronization context present; otherwise they will be subject to the pseudoconcurrency (and simplified thread safety) that we talked about earlier, whereby the only places at which we can be preempted is at an `await` statement. This lets us, for instance, define a shared field, `_x`, and increment it in `GetAnswerToLife` without locking:

```
async Task<int> GetAnswerToLife()
{
  _x++;
  await Task.Delay (5000);
  return 21 * 2;
}
```

(We would, though, be unable to assume that `_x` had the same value before and after the `await`.)

Asynchronous Lambda Expressions

Just as ordinary *named* methods can be asynchronous:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

so can *unnamed* methods (lambda expressions and anonymous methods), if preceded by the `async` keyword:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

We can call and await these in the same way:

```
await NamedMethod();
await unnamed();
```

We can use asynchronous lambda expressions when attaching event handlers:

```
myButton.Click += async (sender, args) =>
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

This is more succinct than the following, which has the same effect:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
```

```
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Asynchronous lambda expressions can also return `Task<TResult>`:

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

Asynchronous Streams (C# 8)

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

Asynchronous streams build on the following pair of interfaces, which are asynchronous counterparts to the enumeration interfaces we described in “Enumeration and Iterators” in Chapter 4:

```
public interface IAsyncEnumerable<out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator (...);
}

public interface IAsyncEnumerator<out T>: IAsyncDisposable
{
  T Current { get; }
  ValueTask<bool> MoveNextAsync();
}
```

`ValueTask<T>` is a struct that wraps `Task<T>` and is behaviorally similar to `Task<T>` while enabling more efficient execution when the task completes synchronously (which can happen often when enumerating a sequence). See "ValueTask<T>" for a discussion of differences. `IAsyncDisposable` is an asynchronous version of `IDisposable`; it provides an opportunity to perform cleanup should you choose to manually implement the interfaces:

```
public interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```

NOTE

The act of fetching each element from the sequence (`MoveNextAsync`) is an asynchronous operation, so asynchronous streams are suitable when elements arrive in a piecemeal fashion (such as when processing data from a video stream). In contrast, the following type is more suitable when the sequence *as a whole* is delayed, but the elements, when they arrive, arrive all together:

```
Task<IEnumerable<T>>
```

To generate an asynchronous stream, you write a method that combines the principles of iterators and asynchronous methods. In other words, your method should include both `yield return` and `await`, and it should return `IAsyncEnumerable<T>`:

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

To consume an asynchronous stream, use the `await foreach` statement:

```
await foreach (var number in RangeAsync (0, 10, 500))
  Console.WriteLine (number);
```

Note that data arrives steadily, every 500 milliseconds (or, in real life, as it becomes available). Contrast this to a similar construct using `Task<IEnumerable<T>>` for which no data is returned until the last piece of data is available:

```
static async Task<IEnumerable<int>> RangeTaskAsync(int start, int count,
                                                               int
delay)
{
  List<int> data = new List<int>();
  for (int i = start; i < start + count; i++)
  {
    await Task.Delay (delay);
    data.Add (i);
  }

  return data;
}
```

Here’s how to consume it with the `foreach` statement:

```
foreach (var data in await RangeTaskAsync(0, 10, 500))
  Console.WriteLine (data);
```

QUERYING IASYNCENUMERABLE<T>

The *System.Linq.Async* NuGet package defines LINQ query operators that operate over `IAsyncEnumerable<T>`, allowing you to write queries much as you would with `IEnumerable<T>`.

For instance, we can write a LINQ query over the `RangeAsync` method that we defined in the preceding section, as follows:

```
IAsyncEnumerable<int> query =
  from i in RangeAsync (0, 10, 500)
  where i % 2 == 0   // Even numbers only.
  select i * 10;     // Multiply by 10.

await foreach (var number in query)
  Console.WriteLine (number);
```

This outputs 0, 20, 40, and so on.

NOTE

If you’re familiar with Reactive Extensions, you can benefit from its (more powerful) query operators, too, by calling the `ToObservable` extension method, which converts an `IAsyncEnumerable<T>` into an `IObservable<T>`. A `ToAsyncEnumerable` extension method is also available, to convert in the reverse direction.

IASYNCENUMERABLE<T> IN ASP.NET CORE

ASP.Net Core controller actions can now return `IAsyncEnumerable<T>`. Such methods must be marked async. For example:

```
[HttpGet]
public async IAsyncEnumerable<string> Get()
{
    using var dbContext = new BookContext();
    await foreach (var title in dbContext.Books
                                         .Select(b => b.Title)
                                         .AsAsyncEnumerable())
       yield return title;
}
```

Asynchronous Methods in WinRT

In WinRT libraries, the equivalent of `Task` is `IAsyncAction` and the equivalent of `Task<TResult>` is `IAsyncOperation<TResult>`. And for operations that report progress, the equivalents are `IAsyncOperationWithProgress<TResult>` and `IAsyncOperationWithProgress<TResult>`. They are all defined in the `Windows.Foundation` namespace.

You can convert from either into a `Task` or `Task<TResult>` via the `AsTask` extension method:

```
Task<StorageFile> fileTask =
KnownFolders.DocumentsLibrary.CreateFileAsync
                              ("test.txt").AsTask();
```

Or, you can await them directly:

```
StorageFile file = await KnownFolders.DocumentsLibrary.CreateFileAsync
                          ("test.txt");
```

NOTE

Due to limitations in the COM type system, `IAsyncOperation<TResult>` and `IAsyncOperationWithProgress<TResult>` are not based on `IAsyncAction` as you might expect. Instead, both inherit from a common base type called `IAsyncInfo`.

The `AsTask` method is also overloaded to accept a cancellation token (see "Cancellation"). It can also accept an `IProgress<T>` object when chained to the `WithProgress` variants (see "Progress Reporting").

Asynchrony and Synchronization Contexts

We've already seen how the presence of a synchronization context is significant in terms of posting continuations. There are a couple of other more subtle ways in which such synchronization contexts come into play with void-returning asynchronous functions. These are not a direct result of C# compiler expansions, but a function of the `Async*MethodBuilder` types in the `System.CompilerServices` namespace that the compiler uses in expanding asynchronous functions.

EXCEPTION POSTING

It's common practice in rich-client applications to rely on the central exception handling event (`Application.DispatcherUnhandledException` in WPF) to process unhandled exceptions thrown on the UI thread. And in ASP.NET Core applications, a custom `ExceptionFilterAttribute` in the `ConfigureServices` method of *Startup.cs* does a similar job. Internally, they work by invoking UI events (or in ASP.NET Core, the pipeline of page-processing methods) in their own `try`/`catch` block.

Top-level asynchronous functions complicate this. Consider the following event handler for a button click:

```
async void ButtonClick (object sender, RoutedEventArgs args)
{
  await Task.Delay(1000);
  throw new Exception ("Will this be ignored?");
}
```

When the button is clicked and the event handler runs, execution returns normally to the message loop after the `await` statement, and the exception that’s thrown a second later cannot be caught by the `catch` block in the message loop.

To mitigate this problem, `AsyncVoidMethodBuilder` catches unhandled exceptions (in void-returning asynchronous functions) and posts them to the synchronization context if present, ensuring that global exception-handling events still fire.

NOTE

The compiler applies this logic only to *void*-returning asynchronous functions. So, if we changed `ButtonClick` to return a `Task` instead of `void`, the unhandled exception would fault the resultant `Task`, which would then have nowhere to go (resulting in an *unobserved* exception).

An interesting nuance is that it makes no difference whether you throw before or after an `await`. Thus, in the following example, the exception is posted to the synchronization context (if present) and never to the caller:

```
async void Foo() { throw null; await Task.Delay(1000); }
```

(If no synchronization context is present, the exception will propagate on the thread pool, which will terminate the application.)

The reason for the exception not being thrown directly back to the caller is to ensure predictability and consistency. In the following example, the `InvalidOperationException` will always have the same effect of faulting the resultant `Task`—regardless of *`someCondition`*:

```
async Task Foo()
{
  if (someCondition) await Task.Delay (100);
  throw new InvalidOperationException();
}
```

Iterators work in a similar way:

```
IEnumerable<int> Foo() { throw null; yield return 123; }
```

In this example, an exception is never thrown straight back to the caller: not until the sequence is enumerated is the exception thrown.

OPERATIONSTARTED AND OPERATIONCOMPLETED

If a synchronization context is present, void-returning asynchronous functions also call its `OperationStarted` method upon entering the function, and its `OperationCompleted` method when the function finishes

Overriding these methods is useful if writing a custom synchronization context for unit testing void-returning asynchronous methods. This is discussed on Microsoft's Parallel Programming blog.

Optimizations

COMPLETING SYNCHRONOUSLY

An asynchronous function can return *before* awaiting. Consider the following method that caches the downloading of web pages:

```
static Dictionary<string,string> _cache = new Dictionary<string,string>
();

async Task<string> GetWebPageAsync (string uri)
{
  string html;
  if (_cache.TryGetValue (uri, out html)) return html;
  return _cache [uri] =
    await new WebClient().DownloadStringTaskAsync (uri);
}
```

Should a URI already exist in the cache, execution returns to the caller with no awaiting having occurred, and the method returns an *already-signaled* task. This is referred to as synchronous completion.

When you await a synchronously completed task, execution does not return to the caller and bounce back via a continuation; instead, it proceeds immediately to the next statement. The compiler implements this optimization by checking the `IsCompleted` property on the awaiter; in other words, whenever you await:

```
Console.WriteLine (await GetWebPageAsync ("http://oreilly.com"));
```

the compiler emits code to short-circuit the continuation in case of synchronization completion:

```
var awaiter = GetWebPageAsync().GetAwaiter();
```

```
if (awaiter.IsCompleted)
  Console.WriteLine (awaiter.GetResult());
else
  awaiter.OnCompleted (() => Console.WriteLine (awaiter.GetResult());
```

NOTE

Awaiting an asynchronous function that returns synchronously still incurs a (very) small overhead—maybe 20 nanoseconds on a 2019-era PC.

In contrast, bouncing to the thread pool introduces the cost of a context switch—perhaps one or two microseconds, and bouncing to a UI message loop, at least 10 times that (much longer if the UI thread is busy).

It’s even legal to write asynchronous methods that *never* await, although the compiler will generate a warning:

```
async Task<string> Foo() { return "abc"; }
```

Such methods can be useful when overriding virtual/abstract methods, if your implementation doesn’t happen to need asynchrony. (An example is `MemoryStream`’s `ReadAsync`/`WriteAsync` methods; see Chapter 15.) Another way to achieve the same result is to use `Task.FromResult`, which returns an already signaled task:

```
Task<string> Foo() { return Task.FromResult ("abc"); }
```

Our `GetWebPageAsync` method is implicitly thread-safe if called from a UI thread, in that you could invoke it several times in succession (thereby initiating multiple concurrent downloads), and no locking is required to protect the cache. If the series of calls were to the same URI, though, we'd end up initiating multiple redundant downloads, all of which would eventually update the same cache entry (the last one winning). Although not erroneous, it would be more efficient if subsequent calls to the same URI could instead (asynchronously) wait upon the result of the in-progress request.

There's an easy way to accomplish this—without resorting to locks or signaling constructs. Instead of a cache of strings, we create a cache of "futures" (`Task<string>`):

```
static Dictionary<string,Task<string>> _cache =
   new Dictionary<string,Task<string>>();

Task<string> GetWebPageAsync (string uri)
{
  if (_cache.TryGetValue (uri, out var downloadTask)) return
downloadTask;
  return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

(Notice that we don't mark the method as `async`, because we're directly returning the task we obtain from calling `WebClient`'s method).

If we call `GetWebPageAsync` repeatedly with the same URI, we're now guaranteed to get the same `Task<string>` object back. (This has the additional benefit of minimizing garbage collection load.) And if the task is complete, awaiting it is cheap, thanks to the compiler optimization that we just discussed.

We could further extend our example to make it thread-safe without the protection of a synchronization context, by locking around the entire method body:

```
lock (_cache)
  if (_cache.TryGetValue (uri, out var downloadTask))
    return downloadTask;
  else
    return _cache [uri] = new WebClient().DownloadStringTaskAsync (uri);
}
```

This works because we're not locking for the duration of downloading a page (which would hurt concurrency); we're locking for the small duration of checking the cache, starting a new task if necessary, and updating the cache with that task.

VALUETASK<T>

We just described how the compiler optimizes an `await` expression on a synchronously completed task—by short-circuiting the continuation and proceeding immediately to the next statement. If the synchronous completion is due to caching, we saw that caching the task itself can provide an elegant and efficient solution.

NOTE

`ValueTask<T>` is intended for micro-optimization scenarios, and you might never need to write methods that return this type. However, it still pays to be aware of the precautions that we outline in the next section because some .NET Core methods return `ValueType<T>`, and `IAsyncEnumerable<T>` makes use of it, too.

It’s not practical, however, to cache the task in all synchronous completion scenarios. Sometimes, a fresh task must be instantiated, and this creates a (tiny) potential inefficiency. This is because `Task` and `Task<T>` are reference types, and so instantiation requires a heap-based memory allocation and subsequent collection. An extreme form of optimization is to write code that’s allocation-free; in other words, that does not instantiate any reference types, adding no burden to garbage collection. To support this pattern, the `ValueTask` and `ValueTask<T>` structs have been introduced, which the compiler allows in place of `Task` and `Task<T>`:

```
async ValueTask<int> Foo() { ... }
```

Awaiting `ValueTask<T>` is allocation-free, *if the operation completes synchronously*:

```
int answer = await Foo();   // (Potentially) allocation-free
```

If the operation doesn’t complete synchronously, `ValueTask<T>` creates an ordinary `Task<T>` behind the scenes (to which it forwards the await), and nothing is gained.

You can convert a `ValueTask<T>` into an ordinary `Task<T>` by calling the `AsTask` method.

There’s also a nongeneric version—`ValueTask`—which is akin to `Task`.

PRECAUTIONS WHEN USING VALUETASK<T>

`ValueTask<T>` is relatively unusual in that it’s defined as a struct *purely* for performance reasons. This means that it’s encumbered with *inappropriate* value-type semantics, which can lead to surprises. To avoid incorrect behavior, you must avoid the following:

- Awaiting the same `ValueTask<T>` multiple times
- Calling `.GetAwaiter().GetResult()` when the operation hasn’t completed

If you need to perform these actions, call `.AsTask()` and operate instead on the resulting `Task`.

NOTE

The easiest way to avoid these traps is to directly await a method call, for instance:

```
await Foo();   // Safe
```

The door to erroneous behavior opens when you assign the (value) task to a variable:

```
ValueTask<int> valueTask = Foo();  // Caution!
// Our use of valueTask can now lead to errors.
```

which can be mitigated by converting immediately to an ordinary task:

```
Task<int> task = Foo().AsTask();   // Safe
// task is safe to work with.
```

AVOIDING EXCESSIVE BOUNCING

For methods that are called many times in a loop, you can avoid the cost of repeatedly bouncing to a UI message loop by calling `ConfigureAwait`. This forces a task not to bounce continuations to the synchronization context, cutting the overhead closer to the cost of a context switch (or much less if the method that you're awaiting completes synchronously):

```
async void A() { ... await B(); ... }
async Task B()
{
  for (int i = 0; i < 1000; i++)
    await C().ConfigureAwait (false);
}

async Task C() { ... }
```

This means that for the `B` and `C` methods, we rescind the simple thread-safety model in UI apps whereby code runs on the UI thread and can be preempted only during an `await` statement. Method `A`, however, is unaffected and will remain on a UI thread if it started on one.

This optimization is particularly relevant when writing libraries: you don’t need the benefit of simplified thread safety because your code typically does not share state with the caller—and does not access UI controls. (It would also make sense, in our example, for method `C` to complete synchronously if it knew the operation was likely to be short-running.)

Asynchronous Patterns

Cancellation

It’s often important to be able to cancel a concurrent operation after it’s started, perhaps in response to a user request. A simple way to implement this is with a cancellation flag, which we could encapsulate by writing a class like this:

```
class CancellationToken
{
  public bool IsCancellationRequested { get; private set; }
  public void Cancel() { IsCancellationRequested = true; }
  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested)
      throw new OperationCanceledException();
  }
}
```

We could then write a cancellable asynchronous method as follows:

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000);
    cancellationToken.ThrowIfCancellationRequested();
  }
}
```

When the caller wants to cancel, it calls `Cancel` on the cancellation token that it passed into `Foo`. This sets `IsCancellationRequested` to `true`, which causes `Foo` to fault a short time later with an `OperationCanceledException` (a predefined exception in the `System` namespace designed for this purpose).

Thread safety aside (we should be locking around reading/writing `IsCancellationRequested`), this pattern is effective and the CLR provides a type called `CancellationToken`, which is very similar to what we've just shown. However, it lacks a `Cancel` method; this method is instead exposed on another type called `CancellationTokenSource`. This separation provides some security: a method that has access only to a `CancellationToken` object can check for but not *initiate* cancellation.

To get a cancellation token, we first instantiate a `CancellationTokenSource`:

```
var cancelSource = new CancellationTokenSource();
```

This exposes a `Token` property, which returns a `CancellationToken`. Hence, we could call our `Foo` method as follows:

```
var cancelSource = new CancellationTokenSource();
Task foo = Foo (cancelSource.Token);
...
... (some time later)
cancelSource.Cancel();
```

Most asynchronous methods in the CLR support cancellation tokens, including `Delay`. If we modify `Foo` such that it passes its token into the `Delay` method, the task will end immediately upon request (rather than up to a second later):

```
async Task Foo (CancellationToken cancellationToken)
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine (i);
    await Task.Delay (1000, cancellationToken);
  }
}
```

Notice that we no longer need to call `ThrowIfCancellationRequested`, because `Task.Delay` is doing that for us. Cancellation tokens propagate nicely down the call stack (just as cancellation requests cascade *up* the call stack, by virtue of being exceptions).

> **NOTE**
>
> Asynchronous methods in WinRT follow an inferior protocol for cancellation whereby instead of accepting a `CancellationToken`, the `IAsyncInfo` type exposes a `Cancel` method. The `AsTask` extension method is overloaded to accept a cancellation token, however, bridging the gap.

Synchronous methods can support cancellation, too (such as `Task`'s `Wait` method). In such cases, the instruction to cancel will need to come asynchronously (e.g., from another task); for example:

```
var cancelSource = new CancellationTokenSource();
Task.Delay (5000).ContinueWith (ant =>
cancelSource.Cancel());
...
```

In fact, you can specify a time interval when constructing `CancellationTokenSource` to initiate cancellation after a set period of time (just as we demonstrated). It's useful for implementing timeouts, whether synchronous or asynchronous:

```
var cancelSource = new CancellationTokenSource (5000);
try { await Foo (cancelSource.Token); }
catch (OperationCanceledException ex) { Console.WriteLine ("Cancelled");
}
```

The `CancellationToken` struct provides a `Register` method that lets you register a callback delegate that will be fired upon cancellation; it returns an object that can be disposed to undo the registration.

Tasks generated by the compiler's asynchronous functions automatically enter a *Canceled* state upon an unhandled `OperationCanceledException` (`IsCanceled` returns true and `IsFaulted` returns false). The same goes for tasks created with `Task.Run` for which you pass the (same) `CancellationToken` to the constructor. The distinction between a faulted and a canceled task is unimportant in asynchronous scenarios, in that both throw an `OperationCanceledException` when awaited; it matters in advanced parallel programming scenarios (specifically conditional continuations). We pick up this topic in "Canceling Tasks".

Progress Reporting

Sometimes, you'll want an asynchronous operation to report back progress as it's running. A simple solution is to pass an `Action` delegate to the asynchronous method, which the method fires whenever progress changes:

```
Task Foo (Action<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged (i / 10);
      // Do something compute-bound...
    }
  });
}
```

Here’s how we could call it:

```
Action<int> progress = i => Console.WriteLine (i + " %");
await Foo (progress);
```

Although this works well in a Console application, it’s not ideal in rich-client scenarios because it reports progress from a worker thread, causing potential thread-safety issues for the consumer. (In effect, we’ve allowed a side effect of concurrency to *leak* to the outside world, which is unfortunate given that the method is otherwise isolated if called from a UI thread.)

IPROGRESS<T> AND PROGRESS<T>

The CLR provides a pair of types to solve this problem: an interface called `IProgress<T>` and a class that implements this interface called `Progress<T>`. Their purpose, in effect, is to *wrap* a delegate so that UI applications can report progress safely through the synchronization context.

The interface defines just one method:

```
public interface IProgress<in T>
{
  void Report (T value);
}
```

Using `IProgress<T>` is easy: our method hardly changes:

```
Task Foo (IProgress<int> onProgressPercentChanged)
{
  return Task.Run (() =>
  {
    for (int i = 0; i < 1000; i++)
    {
      if (i % 10 == 0) onProgressPercentChanged.Report (i / 10);
      // Do something compute-bound...
    }
  });
}
```

The `Progress<T>` class has a constructor that accepts a delegate of type `Action<T>` that it wraps:

```
var progress = new Progress<int> (i => Console.WriteLine (i + "
%"));
await Foo (progress);
```

(`Progress<T>` also has a `ProgressChanged` event that you can subscribe to instead of [or in addition to] passing an action delegate to the constructor.) Upon instantiating `Progress<int>`, the class captures the synchronization context, if present. When `Foo` then calls `Report`, the delegate is invoked through that context.

Asynchronous methods can implement more elaborate progress reporting by replacing `int` with a custom type that exposes a range of properties.

NOTE

If you're familiar with Reactive Framework, you'll notice that `IProgress<T>` together with the task returned by the asynchronous function provide a feature set similar to `IObserver<T>`. The difference is that a task can expose a "final" return value *in addition* to (and differently typed to) the values emitted by `IProgress<T>`.

Values emitted by `IProgress<T>` are typically "throwaway" values (e.g., percent complete or bytes downloaded so far), whereas values pushed by `IObserver<T>`'s `OnNext` typically comprise the result itself and are the very reason for calling it.

Asynchronous methods in WinRT also offer progress reporting, although the protocol is complicated by COM's (relatively) retarded type system. Instead of accepting an `IProgress<T>` object, asynchronous WinRT methods that report progress return one of the following interfaces, in place of `IAsyncAction` and `IAsyncOperation<TResult>`:

```
IAsyncActionWithProgress<TProgress>
IAsyncOperationWithProgress<TResult, TProgress>
```

Interestingly, both are based on `IAsyncInfo` (not `IAsyncAction` and `IAsyncOperation<TResult>`).

The good news is that the `AsTask` extension method is also overloaded to accept `IProgress<T>` for the aforementioned interfaces, so as a .NET consumer, you can ignore the COM interfaces and do this:

```
var progress = new Progress<int> (i => Console.WriteLine (i + " %"));
CancellationToken cancelToken = ...
var task = someWinRTobject.FooAsync().AsTask (cancelToken, progress);
```

The Task-Based Asynchronous Pattern

.NET Core exposes hundreds of task-returning asynchronous methods that you can `await` (mainly related to I/O). Most of these methods (at least partly) follow a pattern called the *Task-Based Asynchronous Pattern* (TAP), which is a sensible formalization of what we have described to date. A TAP method does the following:

- Returns a “hot” (running) `Task` or `Task<TResult>`
- Has an “Async” suffix (except for special cases such as task combinators)
- Is overloaded to accept a cancellation token and/or `IProgress<T>` if it supports cancellation and/or progress reporting
- Returns quickly to the caller (has only a small initial *synchronous phase*)
- Does not tie up a thread if I/O-bound

As we’ve seen, TAP methods are easy to write with C#’s asynchronous functions.

Task Combinators

A nice consequence of there being a consistent protocol for asynchronous functions (whereby they consistently return tasks) is that it’s possible to use and write *task combinators*—functions that usefully combine tasks, without regard for what those specific tasks do.

The CLR includes two task combinators: `Task.WhenAny` and `Task.WhenAll`. In describing them, we’ll assume the following methods are defined:

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }
async Task<int> Delay2() { await Task.Delay (2000); return 2; }
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

WHENANY

`Task.WhenAny` returns a task that completes when any one of a set of tasks complete. The following completes in one second:

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(),
Delay3());
Console.WriteLine ("Done");
Console.WriteLine (winningTask.Result);   // 1
```

Because `Task.WhenAny` itself returns a task, we await it, which returns the task that finished first. Our example is entirely nonblocking—including the last line when we access the `Result` property (because `winningTask` will already have finished). Nonetheless, it's usually better to `await` the `winningTask`:

```
Console.WriteLine (await winningTask);   // 1
```

because any exceptions are then rethrown without an `AggregateException` wrapping. In fact, we can perform both `await`s in one step:

```
int answer = await await Task.WhenAny (Delay1(), Delay2(), Delay3());
```

If a nonwinning task subsequently faults, the exception will go unobserved unless you subsequently await the task (or query its `Exception` property).

`WhenAny` is useful for applying timeouts or cancellation to operations that don't otherwise support it:

```
Task<string> task = SomeAsyncFunc();
Task winner = await (Task.WhenAny (task, Task.Delay(5000)));
if (winner != task) throw new TimeoutException();
string result = await task;   // Unwrap result/re-throw
```

Notice that because in this case we're calling `WhenAny` with differently typed tasks, the winner is reported as a plain `Task` (rather than a `Task<string>`).

WHENALL

`Task.WhenAll` returns a task that completes when *all* of the tasks that you pass to it complete. The following completes after three seconds (and demonstrates the *fork/join* pattern):

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

We could get a similar result by awaiting `task1`, `task2`, and `task3` in turn rather than using `WhenAll`:

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();
await task1; await task2; await task3;
```

The difference (apart from it being less efficient by virtue of requiring three awaits rather than one) is that, should `task1` fault, we'll never get to await `task2`/`task3`, and any of their exceptions will go unobserved.

In contrast, `Task.WhenAll` doesn't complete until all tasks have completed—even when there's a fault. And if there are multiple faults, their exceptions are combined into the task's `AggregateException` (this is when `AggregateException` actually becomes useful—should you be interested in all the exceptions, that is). Awaiting the combined task, however, throws only the first exception, so to see all the exceptions you need to do this:

```
Task task1 = Task.Run (() => { throw null; } );
Task task2 = Task.Run (() => { throw null; } );
Task all = Task.WhenAll (task1, task2);
try { await all; }
```

```
catch
{
  Console.WriteLine (all.Exception.InnerExceptions.Count);   // 2
}
```

Calling `WhenAll` with tasks of type `Task<TResult>` returns a `Task<TResult[]>`, giving the combined results of all the tasks. This reduces to a `TResult[]` when awaited:

```
Task<int> task1 = Task.Run (() => 1);
Task<int> task2 = Task.Run (() => 2);
int[] results = await Task.WhenAll (task1, task2);   // { 1, 2 }
```

To give a practical example, the following downloads URIs in parallel and sums their total length:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<byte[]>> downloadTasks = uris.Select (uri =>
    new WebClient().DownloadDataTaskAsync (uri));

  byte[][] contents = await Task.WhenAll (downloadTasks);
  return contents.Sum (c => c.Length);
}
```

There’s a slight inefficiency here, though, in that we’re unnecessarily hanging on to the byte arrays that we download until every task is complete. It would be more efficient if we collapsed byte arrays into their lengths immediately after downloading them. This is where an asynchronous lambda comes in handy because we need to feed an `await` expression into LINQ’s `Select` query operator:

```
async Task<int> GetTotalSize (string[] uris)
{
  IEnumerable<Task<int>> downloadTasks = uris.Select (async uri =>
    (await new WebClient().DownloadDataTaskAsync (uri)).Length);

  int[] contentLengths = await Task.WhenAll (downloadTasks);
  return contentLengths.Sum();
}
```

CUSTOM COMBINATORS

It can be useful to write your own task combinators. The simplest "combinator" accepts a single task, such as the following, which lets you await any task with a timeout:

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult>
task,
                                                 TimeSpan timeout)
{
  Task winner = await Task.WhenAny (task, Task.Delay (timeout))
                          .ConfigureAwait (false);
  if (winner != task) throw new TimeoutException();
  return await task.ConfigureAwait (false);   // Unwrap result/re-throw
}
```

Because this is very much a "library method" that doesn't access external shared state, we use `ConfigureAwait(false)` when awaiting to avoid potentially bouncing to a UI synchronization context. We can further improve efficiency by canceling the `Task.Delay` when the task completes on time (this avoids the small overhead of a timer hanging around):

```
async static Task<TResult> WithTimeout<TResult> (this Task<TResult>
task,
                                                  TimeSpan timeout)
{
  var cancelSource = new CancellationTokenSource();
  var delay = Task.Delay (timeout, cancelSource.Token);
  Task winner = await Task.WhenAny (task, delay).ConfigureAwait (false);
  if (winner == task)
    cancelSource.Cancel();
  else
    throw new TimeoutException();
  return await task.ConfigureAwait (false);   // Unwrap result/re-throw
}
```

The following lets you “abandon” a task via a `CancellationToken`:

```
static Task<TResult> WithCancellation<TResult> (this Task<TResult> task,
                                          CancellationToken cancelToken)
{
  var tcs = new TaskCompletionSource<TResult>();
  var reg = cancelToken.Register (() => tcs.TrySetCanceled ());
  task.ContinueWith (ant =>
  {
    reg.Dispose();
    if (ant.IsCanceled)
      tcs.TrySetCanceled();
    else if (ant.IsFaulted)
      tcs.TrySetException (ant.Exception.InnerException);
    else
      tcs.TrySetResult (ant.Result);
  });
  return tcs.Task;
}
```

Task combinators can be complex to write, sometimes requiring the use of signaling constructs, which we cover in Chapter 22. This is actually a good thing, because it keeps concurrency-related complexity out of your business logic and into reusable methods that can be tested in isolation.

The next combinator works like `WhenAll`, except that if any of the tasks fault, the resultant task faults immediately:

```
async Task<TResult[]> WhenAllOrError<TResult>
  (params Task<TResult>[] tasks)
{
  var killJoy = new TaskCompletionSource<TResult[]>();
  foreach (var task in tasks)
    task.ContinueWith (ant =>
    {
      if (ant.IsCanceled)
        killJoy.TrySetCanceled();
      else if (ant.IsFaulted)
        killJoy.TrySetException (ant.Exception.InnerException);
    });
  return await await Task.WhenAny (killJoy.Task, Task.WhenAll (tasks))
                         .ConfigureAwait (false);
}
```

We begin by creating a `TaskCompletionSource` whose sole job is to end the party if a task faults. Hence, we never call its `SetResult` method; only its `TrySetCanceled` and `TrySetException` methods. In this case, `ContinueWith` is more convenient than `GetAwaiter().OnCompleted` because we're not accessing the tasks' results and wouldn't want to bounce to a UI thread at that point.

Asynchronous Locking

In “Asynchronous semaphores and locks” in Chapter 22, we describe how to use `SemaphoreSlim` to lock or limit concurrency asynchronously.

Obsolete Patterns

.NET employs other patterns for asynchrony, which precede tasks and asynchronous functions. These are rarely required now that task-based asynchrony has become the dominant pattern.

Asynchronous Programming Model

The oldest pattern is called the Asynchronous Programming Model (APM) and uses a pair of methods starting in “Begin” and “End,” and an interface called `IAsyncResult`. To illustrate, let’s take the `Stream` class in `System.IO` and look at its `Read` method. First, the synchronous version:

```
public int Read (byte[] buffer, int offset, int size);
```

You can probably predict what the *task*-based asynchronous version looks like:

```
public Task<int> ReadAsync (byte[] buffer, int offset, int size);
```

Now let’s examine the APM version:

```
public IAsyncResult BeginRead (byte[] buffer, int offset, int size,
                               AsyncCallback callback, object
state);
public int EndRead (IAsyncResult asyncResult);
```

Calling the `Begin*` method initiates the operation, returning an `IAsyncResult` object, which acts as a token for the asynchronous operation. When the operation completes (or faults), the `AsyncCallback` delegate fires:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Whoever handles this delegate then calls the `End*` method, which provides the operation’s return value as well as rethrowing an exception if the operation faulted.

The APM is not only awkward to use, but also surprisingly difficult to implement correctly. The easiest way to deal with APM methods is to call the `Task.Factory.FromAsync` adapter method, which converts an APM method pair into a `Task`. Internally, it uses a `TaskCompletionSource` to give you a task that’s signaled when an APM operation completes or faults.

The `FromAsync` method requires the following parameters:

- A delegate specifying a `Begin`*`XXX`* method
- A delegate specifying an `End`*`XXX`* method
- Additional arguments that will get passed to these methods

`FromAsync` is overloaded to accept delegate types and arguments that match nearly all the asynchronous method signatures found in .NET Core. For instance, assuming `stream` is a `Stream` and `buffer` is a `byte[]`, we could do this:

```
Task<int> readChunk = Task<int>.Factory.FromAsync (
  stream.BeginRead, stream.EndRead, buffer, 0, 1000, null);
```

Event-Based Asynchronous Pattern

The *Event-Based Asynchronous Pattern* (EAP) was introduced in Framework 2.0 to provide a simpler alternative to the APM, particularly in UI scenarios. It was implemented in only a handful of types, however, most notably `WebClient` in `System.Net`. The EAP is just a pattern; no types are provided to assist. Essentially the pattern is this: a class offers a family of members that internally manage concurrency, similar to the following:

```
// These members are from the WebClient class:

public byte[] DownloadData (Uri address);    // Synchronous version
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler
DownloadDataCompleted;

public void CancelAsync (object userState);  // Cancels an
operation
public bool IsBusy { get; }                  // Indicates if
still running
```

The `*Async` methods initiate an operation asynchronously. When the operation completes, the `*Completed` event fires (automatically posting to the captured synchronization context if present). This event passes back an event arguments object that contains the following:

- A flag indicating whether the operation was canceled (by the consumer calling `CancelAsync`)
- An `Error` object indicating an exception that was thrown (if any)
- The `userToken` object if supplied when calling the `Async` method

EAP types can also expose a progress reporting event, which fires whenever progress changes (also posted through the synchronization context):

```
public event DownloadProgressChangedEventHandler
DownloadProgressChanged;
```

Implementing the EAP requires a large amount of boilerplate code, making the pattern poorly compositional.

BackgroundWorker

`BackgroundWorker` in `System.ComponentModel` is a general-purpose implementation of the EAP. It allows rich-client apps to start a worker thread and report completion and percentage-based progress without needing to explicitly capture synchronization context. Here's an example:

```
var worker = new BackgroundWorker { WorkerSupportsCancellation = true };
worker.DoWork += (sender, args) =>
{                                      // This runs on a worker thread
  if (args.Cancel) return;
  Thread.Sleep(1000);
  args.Result = 123;
};
worker.RunWorkerCompleted += (sender, args) =>
{                                                  // Runs on UI thread
  // We can safely update UI controls here...
  if (args.Cancelled)
    Console.WriteLine ("Cancelled");
  else if (args.Error != null)
    Console.WriteLine ("Error: " + args.Error.Message);
  else
    Console.WriteLine ("Result is: " + args.Result);
};
worker.RunWorkerAsync();   // Captures sync context and starts operation
```

`RunWorkerAsync` starts the operation, firing the `DoWork` event on a pooled worker thread. It also captures the synchronization context, and when the operation completes (or faults), the `RunWorkerCompleted` event is invoked through that synchronization context (like a continuation).

`BackgroundWorker` creates coarse-grained concurrency, in that the `DoWork` event runs entirely on a worker thread. If you need to update UI controls in that event handler (other than posting a percentage-complete message), you must use `Dispatcher.BeginInvoke` or similar.

We describe `BackgroundWorker` in more detail online.

1 The CLR creates other threads behind the scenes for garbage collection and finalization.