The Task-based Asynchronous Pattern

Stephen Toub, Microsoft
February 2012

Contents

Overview

The Task-based Asynchronous Pattern (TAP) is a new pattern for asynchrony in the .NET Framework. It is based on the Task and Task<TResult> types in the System.Threading.Tasks namespace, which are used to represent arbitrary asynchronous operations.

The Task-based Asynchronous Pattern Defined

Naming, Parameters, and Return Types

Initiation and completion of an asynchronous operation in the TAP are represented by a single method, and thus there is only one method to name. This is in contrast to the IAsyncResult pattern, or APM pattern, where BeginMethodName and EndMethodName methods are required, and in contrast to the event-based asynchronous pattern, or EAP, where a MethodNameAsync is required in addition to one or more events, event handler delegate types, and EventArg-derived types. Asynchronous methods in the TAP are named with an "Async" suffix that follows the operation's name, e.g. MethodNameAsync. The singular TAP method returns either a Task or a Task<TResult>, based on whether the corresponding synchronous method would return void or a type TResult, respectively. (If adding a TAP method to a class that already contains a method MethodNameAsync, the suffix "TaskAsync" may be used instead, resulting in "MethodNameTaskAsync".)

For example, consider a "Read" method that reads a specified amount of data into a provided buffer starting at a specified offset:

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

The APM counterpart to this method would expose the following two methods:

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

The EAP counterpart would expose the following set of types and members:

```
public class MyClass
{
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}

public delegate void ReadCompletedEventHandler(
    object sender, ReadCompletedEventArgs eventArgs);
```

```
public class ReadCompletedEventArgs : AsyncCompletedEventArgs
{
    public int Result { get; }
}
```

The TAP counterpart would expose the following single method:

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

The parameters to a basic TAP method should be the same parameters provided to the synchronous counterpart, in the same order. However, “out” and “ref” parameters are exempted from this rule and should be avoided entirely. Any data that would have been returned through an out or ref parameter should instead be returned as part of the returned Task<TResult>’s Result, utilizing a tuple or a custom data structure in order to accommodate multiple values.

Methods devoted purely to the creation, manipulation, or combination of tasks (where the asynchronous intent of the method is clear in the method name or in the name of the type on which the method lives) need not follow the aforementioned naming pattern; such methods are often referred to as “combinators.” Examples of such methods include Task.WhenAll and Task.WhenAny, and are discussed in more depth later in this document.

Behavior

Initiating the Asynchronous Operation

An asynchronous method based on the TAP is permitted to do a small amount of work synchronously before it returns the resulting Task. The work should be kept to the minimal amount necessary, performing operations such as validating arguments and initiating the asynchronous operation. It is likely that asynchronous methods will be invoked from user interface threads, and thus any long-running work in the synchronous up-front portion of an asynchronous method could harm responsiveness. It is also likely that multiple asynchronous methods will be launched concurrently, and thus any long-running work in the synchronous up-front portion of an asynchronous method could delay the initiation of other asynchronous operations, thereby decreasing benefits of concurrency.

In some cases, the amount of work required to complete the operation is less than the amount of work it would take to launch the operation asynchronously (e.g. reading from a stream where the read can be satisfied by data already buffered in memory). In such cases, the operation may complete synchronously, returning a Task that has already been completed.

Exceptions

An asynchronous method should only directly raise an exception to be thrown out of the MethodNameAsync call in response to a usage error*. For all other errors, exceptions occurring during the execution of an asynchronous method should be assigned to the returned Task. This is the case even

if the asynchronous method happens to complete synchronously before the Task is returned. Typically, a Task will contain at most one exception. However, for cases where a Task is used to represent multiple operations (e.g. Task.WhenAll), multiple exceptions may be associated with a single Task.

(*Per .NET design guidelines, a usage error is something that can be avoided by changing the code that calls the method. For example, if an error state results when a null is passed as one of the method's arguments, an error condition usually represented by an ArgumentNullException, the calling code can be modified by the developer to ensure that null is never passed. In other words, the developer can and should ensure that usage errors never occur in production code.)

Target Environment

It is up to the TAP method's implementation to determine where asynchronous execution occurs. The developer of the TAP method may choose to execute the workload on the ThreadPool, may choose to implement it using asynchronous I/O and thus without being bound to a thread for the majority of the operation's execution, may choose to run on a specific thread as need-be, such as the UI thread, or any number of other potential contexts. It may even be the case that a TAP method has no execution to perform, returning a Task that simply represents the occurrence of a condition elsewhere in the system (e.g. a Task<TData> that represents TData arriving at a queued data structure).

The caller of the TAP method may block waiting for the TAP method to complete (by synchronously waiting on the resulting Task), or may utilize a continuation to execute additional code when the asynchronous operation completes. The creator of the continuation has control over where that continuation code executes. These continuations may be created either explicitly through methods on the Task class (e.g. ContinueWith) or implicitly using language support built on top of continuations (e.g. "await" in C#, "Await" in Visual Basic, "AwaitValue" in F#).

Task Status

The Task class provides a life cycle for asynchronous operations, and that cycle is represented by the TaskStatus enumeration. In order to support corner cases of types deriving from Task and Task<TResult> as well as the separation of construction from scheduling, the Task class exposes a Start method. Tasks created by its public constructors are referred to as "cold" tasks, in that they begin their life cycle in the non-scheduled TaskStatus.Created state, and it's not until Start is called on these instances that they progress to being scheduled. All other tasks begin their life cycle in a "hot" state, meaning that the asynchronous operations they represent have already been initiated and their TaskStatus is an enumeration value other than Created.

All tasks returned from TAP methods must be "hot." If a TAP method internally uses a Task's constructor to instantiate the task to be returned, the TAP method must call Start on the Task object prior to returning it. Consumers of a TAP method may safely assume that the returned task is "hot," and should not attempt to call Start on any Task returned from a TAP method. Calling Start on a "hot" task will result in an InvalidOperationException (this check is handled automatically by the Task class).

Optional: Cancellation

Cancellation in the TAP is opt-in for both asynchronous method implementers and asynchronous method consumers. If an operation is built to be cancelable, it will expose an overload of the MethodNameAsync method that accepts a System.Threading.CancellationToken. The asynchronous operation will monitor this token for cancellation requests, and if a cancellation request is received, may choose to honor that request and cancel the operation. If the cancellation request is honored such that work is ended prematurely, the Task returned from the TAP method will end in the TaskStatus.Canceled state.

To expose a cancelable asynchronous operation, a TAP implementation provides an overload that accepts a CancellationToken after the synchronous counterpart method's parameters. By convention, the parameter should be named "cancellationToken".

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

If the token has cancellation requested and the asynchronous operation is able to respect that request, the returned task will end in the TaskStatus.Canceled state; there will be no available Result and no Exception. The Canceled state is considered to be a final, or completed, state for a task, along with the Faulted and RanToCompletion states. Thus, a task in the Canceled state will have its IsCompleted property returning true. When a task completes in the Canceled state, any continuations registered with the task will be scheduled or executed, unless such continuations opted out at the time they were created, through use of specific TaskContinuationOptions (e.g. TaskContinuationOptions.NotOnCanceled). Any code asynchronously waiting for a canceled task through use of language features will continue execution and receive an OperationCanceledException (or a type derived from it). Any code blocked synchronously waiting on the task (through methods like Wait or WaitAll) will similarly continue execution with an exception.

If a CancellationToken has cancellation requested prior to the invocation of a TAP method that accepts that token, the TAP method should return a Canceled task. However, if cancellation is requested during the asynchronous operation's execution, the asynchronous operation need not respect the cancellation request. Only if the operation completes due to the cancellation request should the returned Task end in the Canceled state; if cancellation is requested but a result or an exception is still produced, the Task should end in the RanToCompletion or Faulted state, respectively.

For methods that desire having cancellation first and foremost in the mind of a developer using the asynchronous method, an overload need not be provided that doesn't accept a CancellationToken. For methods that are not cancelable, overloads accepting CancellationToken should not be provided; this helps indicate to the caller whether the target method is actually cancelable. A consumer that does not desire cancellation may call a method that accepts a CancellationToken and provide CancellationToken.None as the argument value; CancellationToken.None is functionally equivalent to default(CancellationToken).

Optional: Progress Reporting

Some asynchronous operations benefit from providing progress notifications; these are typically utilized to update a user interface with information about the progress of the asynchronous operation.

In the TAP, progress is handled through an IProgress<T> interface (described later in this document) passed into the asynchronous method as a parameter named "progress". Providing the progress interface at the time of the asynchronous method's invocation helps to eliminate race conditions that result from incorrect usage where event handlers incorrectly registered after the invocation of the operation may miss updates. More importantly, it enables varying implementations of progress to be utilized, as determined by the consumer. The consumer may, for example, only care about the latest progress update, or may want to buffer them all, or may simply want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread; all of this may be achieved by utilizing a different implementation of the interface, each of which may be customized to the particular consumer's need. As with cancellation, TAP implementations should only provide an IProgress<T> parameter if the API supports progress notifications.

For example, if our aforementioned ReadAsync method was able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an IProgress<int>:

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

If a FindFilesAsync method returned a list of all files that met a particular search pattern, the progress callback could provide an estimation as to the percentage of work completed as well as the current set of partial results. It could do this either with a tuple, e.g.:

```
public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
    string pattern,
    IProgress<Tuple<double,ReadOnlyCollection<List<FileInfo>>>> progress);
```

or with a data type specific to the API, e.g.:

```
public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
    string pattern,
    IProgress<FindFilesProgressInfo> progress);
```

In the latter case, the special data type should be suffixed with "ProgressInfo".

If TAP implementations provide overloads that accept a progress parameter, they must allow the argument to be null, in which case no progress will be reported. TAP implementations should synchronously report the progress to the IProgress<T> object, making it cheap for the async implementation to quickly provide progress, and allowing the consumer of the progress to determine how and where best to handle the information (e.g. the progress instance itself could choose to marshal callbacks and raise events on a captured synchronization context).

IProgress<T> Implementations

A single IProgress<T> implementation, Progress<T>, is provided as part of the .NET Framework 4.5 (more implementations may be provided in the future). The Progress<T> class is declared as follows:

```
public class Progress<T> : IProgress<T>
{
    public Progress();
    public Progress(Action<T> handler);
    protected virtual void OnReport(T value);
    public event EventHandler<T> ProgressChanged;
}
```

An instance of Progress<T> exposes a ProgressChanged event, which is raised every time the asynchronous operation reports a progress update. The ProgressChanged event is raised on whatever SynchronizationContext was captured when the Progress<T> instance was instantiated (if no context was available, a default context is used, targeting the ThreadPool). Handlers may be registered with this event; a single handler may also be provided to the Progress instance's constructor (this is purely for convenience, and behaves just as would an event handler for the ProgressChanged event). Progress updates are raised asynchronously so as to avoid delaying the asynchronous operation while event handlers are executing. Another IProgress<T> implementation could choose to apply different semantics.

Choosing Which Overloads to Provide

With both the optional CancellationToken and optional IProgress<T> parameters, an implementation of the TAP could potentially demand up to four overloads:

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…, CancellationToken cancellationToken);
public Task MethodNameAsync(…, IProgress<T> progress);
public Task MethodNameAsync(…,
    CancellationToken cancellationToken, IProgress<T> progress);
```

However, many TAP implementations will have need for only the shortest overload, as they will not provide either cancellation or progress capabilities:

```
public Task MethodNameAsync(…);
```

If an implementation supports either cancellation or progress but not both, a TAP implementation may provide two overloads:

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…, CancellationToken cancellationToken);

// … or …

public Task MethodNameAsync(…);
public Task MethodNameAsync(…, IProgress<T> progress);
```

If an implementation supports both cancellation and progress, it may expose all four potential overloads. However, it is valid to provide only two:

```
public Task MethodNameAsync(…);
public Task MethodNameAsync(…,
    CancellationToken cancellationToken, IProgress<T> progress);
```

To make up for the missing two intermediate combinations, developers may pass CancellationToken.None (or default(CancellationToken)) for the cancellationToken parameter and/or null for the progress parameter.

If it is expected that every usage of the TAP method should utilize cancellation and/or progress, the overloads that don't accept the relevant parameter may be omitted.

If multiple overloads of a TAP method are exposed to make cancellation and/or progress optional, the overloads that don't support cancellation and/or progress should behave as if they'd passed CancellationToken.None for cancellation and null for progress to the overload that does support these.

Implementing the Task-based Asynchronous Pattern

Method Generation

Compiler

In the .NET Framework 4.5, the C# and Visual Basic compilers are capable of implementing the TAP. Any method attributed with the async keyword (Async in Visual Basic) is considered to be an asynchronous method, and the compiler will perform the necessary transformations to implement the method asynchronously using the TAP. Such a method should return either a Task or a Task<TResult>. In the case of the latter, the body of the function should return a TResult, and the compiler will ensure that this result is made available through the resulting Task<TResult>. Similarly, any exceptions that go unhandled within the body of the method will be marshaled to the output task, causing the resulting Task to end in the Faulted state; the one exception to this is if an OperationCanceledException (or derived type) goes unhandled, such that the resulting Task will end in the Canceled state.

Manual

Developers may implement the TAP manually, just as the compiler does or with greater control over exactly how the method is implemented. The compiler relies on public surface area exposed from the System.Threading.Tasks namespace (and supporting types in the System.Runtime.CompilerServices namespace built on top of System.Threading.Tasks), functionality also available to developers directly. For more information, see the following section on Workloads. When implementing a TAP method manually, a developer must be sure to complete the resulting Task when the represented asynchronous operation completes.

Hybrid

It is often useful to manually implement the TAP pattern with the core logic for the implementation implemented in a compiler-generated implementation. This is the case, for example, when arguments should be verified outside of a compiler-generated asynchronous method in order for the exceptions to escape to the method's direct caller rather than being exposed through the Task, e.g.

```
public Task<int> MethodAsync(string input)
{
    if (input == null) throw new ArgumentNullException("input");
    return MethodAsyncInternal(input);
}

private async Task<int> MethodAsyncInternal(string input)
{
    … // code that uses await
}
```

Another case where such delegation is useful is when a "fast path" optimization can be implemented that returns a cached task.

Workloads

Both compute-bound and I/O-bound asynchronous operations may be implemented as TAP methods. However, when exposed publicly from a library, TAP implementations should only be provided for workloads that involve I/O-bound operations (they may also involve computation, but should not be purely computation). If a method is purely compute-bound, it should be exposed only as a synchronous implementation; a consumer may then choose whether to wrap an invocation of that synchronous method into a Task for their own purposes of offloading the work to another thread and/or to achieve parallelism.

Compute-Bound

The Task class is ideally suited to representing computationally-intensive operations. By default, it utilizes special support within the .NET ThreadPool in order to provide efficient execution, while also providing a great deal of control over when, where, and how asynchronous computations execute.

There are several ways compute-bound tasks may be generated.

- In .NET 4, the primary way for launching a new compute-bound task is the TaskFactory.StartNew method, which accepts a delegate (typically an Action or a Func<TResult>) to be executed asynchronously. If an Action is provided, a Task is returned to represent the asynchronous execution of that delegate. If a Func<TResult> is provided, a Task<TResult> is returned. Overloads of StartNew exist that accept CancellationToken, TaskCreationOptions, and TaskScheduler, all of which provide fine-grained control over the task's scheduling and execution. A factory instance that targets the current task scheduler is available as a static property off of the Task class, e.g. Task.Factory.StartNew(…).
- In .NET 4.5, the Task type exposes a static Run method as a shortcut to StartNew and which may be used to easily launch a compute-bound task that targets the ThreadPool. As of .NET 4.5, this

is the preferred mechanism for launching a compute-bound task; StartNew should only be used directly when more fine-grained control is required over its behavior.

- The Task type exposes constructors and a Start method. These may be used if construction must be done separate from scheduling. (As previously mentioned in this document, public APIs must only return tasks that have already been started.)
- The Task type exposes multiple overloads of ContinueWith. This method creates a new task that will be scheduled when another task completes. Overloads exist that accept CancellationToken, TaskContinuationOptions, and TaskScheduler, all of which provide fine-grained control over the continuation task's scheduling and execution.
- The TaskFactory class provides ContinueWhenAll and ContinueWhenAny methods. These methods create a new task that will be scheduled when all of or any of a supplied set of tasks completes. As with ContinueWith, support exists for controlling the scheduling and execution of these tasks.

Compute-bound tasks are special with regards to cancellation, as the system can prevent actually executing a scheduled task if a cancellation request is received prior to the execution starting. As such, if a CancellationToken is provided, in addition to possibly passing that token into the asynchronous code which may monitor the token, the token should also be provided to one of the previously mentioned routines (e.g. StartNew, Run) so that the Task runtime may also monitor the token.

Consider an asynchronous method that renders an image. The body of the task can poll the cancellation token such that, while rendering is occurring, the code may exit early if a cancellation request arrives. In addition, we also want to prevent doing any rendering if the cancellation request occurs prior to rendering starting:

```
public Task<Bitmap> RenderAsync(
    ImageData data, CancellationToken cancellationToken)
{
    return Task.Run(() =>
    {
        var bmp = new Bitmap(data.Width, data.Height);
        for(int y=0; y<data.Height; y++)
        {
            cancellationToken.ThrowIfCancellationRequested();
            for(int x=0; x<data.Width; x++)
            {
                … // render pixel [x,y] into bmp
            }
        }
        return bmp;
    }, cancellationToken);
}
```

Compute-bound tasks will end in a Canceled state if at least one of the following conditions is true:

- The CancellationToken provided as an argument to the creation method (e.g. StartNew, Run) has cancellation requested prior to the Task transitioning to the TaskStatus.Running state.

- An OperationCanceledException goes unhandled within the body of such a Task, that OperationCanceledException contains as its CancellationToken property the same CancellationToken passed into the Task, and that CancellationToken has had cancellation requested.

If another exception goes unhandled within the body of the Task, that Task will end in the Faulted state, and any attempts to wait on the task or access its result will result in an exception being thrown.

I/O-Bound

Tasks that should not be directly backed by a thread for the entirety of their execution are created using the TaskCompletionSource<TResult> type. TaskCompletionSource<TResult> exposes a Task property which returns an associated Task<TResult> instance. The life-cycle of this task is controlled by methods exposed from the TaskCompletionSource<TResult> instance, namely SetResult, SetException, SetCanceled, and their TrySet* variants.

Consider the need to create a task that will complete after a specified duration of time. This could, for example, be useful in UI scenarios where the developer wants to delay an activity for a period of time. The .NET System.Threading.Timer class already provides the ability to asynchronously invoke a delegate after a specified period of time, and using TaskCompletionSource<TResult> we can put a Task façade on the timer, e.g.

```
public static Task<DateTimeOffset> Delay(int millisecondsTimeout)
{
    var tcs = new TaskCompletionSource<DateTimeOffset>();
    new Timer(self =>
    {
        ((IDisposable)self).Dispose();
        tcs.TrySetResult(DateTimeOffset.UtcNow);
    }).Change(millisecondsTimeout, -1);
    return tcs.Task;
}
```

In the .NET Framework 4.5, the Task.Delay method is provided for this purpose. Such a method could now be used inside of another asynchronous method to, for example, implement an asynchronous polling loop:

```
public static async Task Poll(
    Uri url,
    CancellationToken cancellationToken,
    IProgress<bool> progress)
{
    while(true)
    {
        await Task.Delay(TimeSpan.FromSeconds(10), cancellationToken);
        bool success = false;
        try
        {
            await DownloadStringAsync(url);
            success = true;
```

```
            }
            catch { /* ignore errors */ }
            progress.Report(success);
        }
    }
```

There is no non-generic counterpart to TaskCompletionSource<TResult>. However, Task<TResult> derives from Task, and thus the generic TaskCompletionSource<TResult> can be used for I/O-bound methods that simply return a Task by utilizing a source with a dummy TResult (Boolean is a good default choice, and if a developer is concerned about a consumer of the Task downcasting it to a Task<TResult>, a private TResult type may be used). For example, the previously shown Delay method was developed to return the current time along with the resulting Task<DateTimeOffset>. If such a result value is unnecessary, the method could have instead been coded as follows (note the change of return type and the change of argument to TrySetResult):

```
public static Task Delay(int millisecondsTimeout)
{
    var tcs = new TaskCompletionSource<bool>();
    new Timer(self =>
    {
        ((IDisposable)self).Dispose();
        tcs.TrySetResult(true);
    }).Change(millisecondsTimeout, -1);
    return tcs.Task;
}
```

Mixed Compute- and I/O-bound Tasks

Asynchronous methods are not limited to just compute-bound or I/O-bound operations, but may represent a mixture of the two. In fact, it is often the case that multiple asynchronous operations of different natures are composed together into larger mixed operations. For example, consider the previously shown RenderAsync method which performed a computationally-intensive operation to render an image based on some input ImageData. This ImageData could come from a Web service which we asynchronously access:

```
public async Task<Bitmap> DownloadDataAndRenderImageAsync(
    CancellationToken cancellationToken)
{
    var imageData = await DownloadImageDataAsync(cancellationToken);
    return await RenderAsync(imageData, cancellationToken);
}
```

This example also demonstrates how a single CancellationToken may be threaded through multiple asynchronous operations. More on this topic is discussed in the cancellation usage section later in this document.

Consuming the Task-based Asynchronous Pattern

Await

At the API level, the way to achieve waiting without blocking is to provide callbacks. For Tasks, this is achieved through methods like ContinueWith. Language-based asynchrony support hides callbacks by allowing asynchronous operations to be awaited within normal control flow, with compiler-generated code targeting this same API-level support.

In .NET 4.5, C# and Visual Basic directly support asynchronously awaiting Task and Task<TResult>, with the 'await' keyword in C# and the 'Await' keyword in Visual Basic. If awaiting a Task, the await expression is of type void; if awaiting a Task<TResult>, the await expression is of type TResult. An await expression must occur inside the body of an asynchronous method, with return type void, Task, or Task<TResult> for some TResult. For more information on the C# and Visual Basic language support in the .NET Framework 4.5, see the C# and Visual Basic language specifications.

Under the covers, the await functionality installs a callback on the task via a continuation. This callback will resume the asynchronous method at the point of suspension. When the asynchronous method is resumed, if the awaited operation completed successfully and was a Task<TResult>, its TResult will be returned. If the Task or Task<TResult> awaited ended in the Canceled state, an OperationCanceledException will be thrown. If the Task or Task<TResult> awaited ended in the Faulted state, the exception that caused it to fault will be thrown. It is possible for a Task to fault due to multiple exceptions, in which case only one of these exceptions will be propagated; however, the Task's Exception property will return an AggregateException containing all of the errors.

If a SynchronizationContext is associated with the thread executing the asynchronous method at the time of suspension (e.g. SynchronizationContext.Current is non-null), the resumption of the asynchronous method will take place on that same SynchronizationContext through usage of the context's Post method. Otherwise, it will rely on whatever System.Threading.Tasks.TaskScheduler was current at the time of suspension (typically this will be TaskScheduler.Default, which targets the .NET ThreadPool). It's up to this TaskScheduler whether to allow the resumption to execute wherever the awaited asynchronous operation completed or to force the resumption to be scheduled. The default scheduler will typically allow the continuation to run on whatever thread the awaited operation completed.

When called, an asynchronous method synchronously executes the body of the function up until the first await expression on an awaitable instance that is not yet completed, at which point the invocation returns to the caller. If the asynchronous method does not return void, a Task or Task<TResult> is returned to represent the ongoing computation. In a non-void asynchronous method, if a return statement is encountered, or the end of the method body is reached, the task is completed in the RanToCompletion final state. If an unhandled exception causes control to leave the body of the asynchronous method, the task ends in the Faulted state (if that exception is an OperationCanceledException, the task instead ends in the Canceled state). In this manner, the result or exception will eventually be published.

There are several important variations from the described behavior. For performance reasons, if a task has already completed by the time the task is awaited, control will not be yielded, and the function will instead continue executing. Additionally, it is not always the desired behavior to return back to the original context; method-level support is provided to change this behavior, and this is described in more detail later in this document.

Yield and ConfigureAwait

Several members give more control over an asynchronous method's execution. The Task class provides a Yield method that may be used to introduce a yield point into the asynchronous method.

```
public class Task : …
{
    public static YieldAwaitable Yield();
    …
}
```

This is equivalent to asynchronously posting or scheduling back to whatever context is current.

```
Task.Run(async delegate
{
    for(int i=0; i<1000000; i++)
    {
        await Task.Yield(); // fork the continuation into a separate work item
        ...
    }
});
```

The Task class also provides a ConfigureAwait method which gives more control over how suspension and resumption occur in an asynchronous method. As mentioned previously, by default the current context at the time an async method is suspended is captured, and that captured context is used to invoke the async method's continuation upon resumption. In many cases, this is the exact behavior you want. However, in some cases you don't care where you end up, and as a result you can achieve better performance by avoiding such posts back to the original context. To enable this, ConfigureAwait can be used to inform the await operation not to capture and resume on the context, instead preferring to continue execution wherever the asynchronous operation being awaited completed:

```
await someTask.ConfigureAwait(continueOnCapturedContext:false);
```

Cancellation

TAP methods that are cancelable expose at least one overload that accepts a CancellationToken, a type introduced to System.Threading in .NET 4.

A CancellationToken is created through a CancellationTokenSource. The source's Token property returns the CancellationToken that will be signaled when the source's Cancel method is invoked. For example, consider downloading a single Web page and wanting to be able to cancel the operation. We create a CancellationTokenSource, pass its token to the TAP method, and later potentially call the source's Cancel method:

```
var cts = new CancellationTokenSource();
string result = await DownloadStringAsync(url, cts.Token);
… // at some point later, potentially on another thread
cts.Cancel();
```

To cancel multiple asynchronous invocations, the same token may be passed into all invocations:

```
var cts = new CancellationTokenSource();
IList<string> results = await Task.WhenAll(
    from url in urls select DownloadStringAsync(url, cts.Token));
…
cts.Cancel();
```

Similarly, the same token may be selectively passed to only a subset of operations:

```
var cts = new CancellationTokenSource();
byte [] data = await DownloadDataAsync(url, cts.Token);
await SaveToDiskAsync(outputPath, data, CancellationToken.None);
…
cts.Cancel();
```

Cancellation requests may be initiated from any thread.

CancellationToken.None may be passed to any method accepting a CancellationToken in order to indicate that cancellation will never be requested. The callee will find that the cancellationToken's CanBeCanceled will return false, and the callee can optimize accordingly. (For testing purposes, a pre-canceled CancellationToken may also be passed in, constructed using CancellationToken's constructor that accepts a Boolean value to indicate whether the token should start in an already-canceled or not-cancelable state.)

The same CancellationToken may be handed out to any number of asynchronous and synchronous operations. This is one of the strong suits of the CancellationToken approach: cancellation may be requested of synchronous method invocations, and the same cancellation request may be proliferated to any number of listeners. Another benefit of this approach is that the developer of the asynchronous API is in complete control of whether cancellation may be requested and of when cancellation may take effect, and the consumer of the API may selectively determine to which of multiple asynchronous invocations cancellation requests will be propagated.

Progress

Some asynchronous methods expose progress through a progress interface passed into the asynchronous method. For example, consider a function which asynchronously downloads a string of text, and along the way raises progress updates that include the percentage of the download that has completed thus far. Such a method could be consumed in a Windows Presentation Foundation application as follows:

```
private async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
```

```
        {
            txtResult.Text = await DownloadStringAsync(txtUrl.Text,
                new Progress<int>(p => pbDownloadProgress.Value = p));
        }
        finally { btnDownload.IsEnabled = true; }
    }
```

Using the Built-in Task-based Combinators

The System.Threading.Tasks namespace includes several key methods for working with and composing tasks.

Task.Run

The Task class exposes several Run methods that enable easily offloading work as a Task or Task<TResult> to the ThreadPool, e.g.

```
    public async void button1_Click(object sender, EventArgs e)
    {
        textBox1.Text = await Task.Run(() =>
        {
            // … do compute-bound work here
            return answer;
        });
    }
```

Some of these run methods (such as the `Run<TResult>(Func<TResult>)` overload used in the prior snippet) exist as shorthand for the TaskFactory.StartNew method that's existed since .NET 4. Other overloads, however, such as `Run<TResult>(Func<Task<TResult>>)`, enable await to be used within the offloaded work, e.g.

```
    public async void button1_Click(object sender, EventArgs e)
    {
        pictureBox1.Image = await Task.Run(() =>
        {
            using(Bitmap bmp1 = await DownloadFirstImageAsync())
            using(Bitmap bmp2 = await DownloadSecondImageAsync())
            return Mashup(bmp1, bmp2);
        });
    }
```

Such overloads are logically equivalent to using StartNew in conjunction with the Unwrap extension method in the Task Parallel Library.

Task.FromResult

For scenarios where data may already be available and simply needs to be returned from a task-returning method lifted into a Task<TResult>, the Task.FromResult method may be used:

```
    public Task<int> GetValueAsync(string key)
    {
        int cachedValue;
        return TryGetCachedValue(out cachedValue) ?
            Task.FromResult(cachedValue) :
```

```
        GetValueAsyncInternal();
}

private async Task<int> GetValueAsyncInternal(string key)
{
    …
}
```

Task.WhenAll

The WhenAll method is used to asynchronously wait on multiple asynchronous operations represented as Tasks. It has multiple overloads in order to accommodate a set of non-generic tasks or a non-uniform set of generic tasks (e.g. asynchronously waiting for multiple void-returning operations, or asynchronously waiting for multiple value-returning methods where each of the values may be of a different type) as well as a uniform set of generic tasks (e.g. asynchronously waiting for multiple TResult-returning methods).

Consider the need to send emails to several customers. We can overlap the sending of all of the emails (there's no need to wait for one email to complete sending before sending the next), and we need to know when the sends have completed and if any errors occurred:

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
await Task.WhenAll(asyncOps);
```

The above code does not explicitly handle exceptions that may occur, instead choosing to let exceptions propagate out of the await on WhenAll's resulting task. To handle the exceptions, the developer could employ code like the following:

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
try
{
    await Task.WhenAll(asyncOps);
}
catch(Exception exc)
{
    ...
}
```

In the previous case, if any of the asynchronous operations failed, all of the exceptions will be gathered up into an AggregateException which will be stored in the Task returned from WhenAll. However, only one of those exceptions will be propagated by the await keyword. If it is important to be able to examine all of the exceptions, the above snippet may be rewritten as follows:

```
Task [] asyncOps = (from addr in addrs select SendMailAsync(addr)).ToArray();
try
{
    await Task.WhenAll(asyncOps);
}
catch(Exception exc)
{
```

```
    foreach(Task faulted in asyncOps.Where(t => t.IsFaulted))
    {
        … // work with faulted and faulted.Exception
    }
}
```

Consider another example of downloading multiple files from the web asynchronously. In this case, all of the asynchronous operations have homogeneous result types, and access to the results is simple:

```
string [] pages = await Task.WhenAll(
    from url in urls select DownloadStringAsync(url));
```

As in the previous void-returning case, the same exception handling techniques are usable here:

```
Task [] asyncOps =
    (from url in urls select DownloadStringAsync(url)).ToArray();
try
{
    string [] pages = await Task.WhenAll(asyncOps);
    ...
}
catch(Exception exc)
{
    foreach(Task<string> faulted in asyncOps.Where(t => t.IsFaulted))
    {
        … // work with faulted and faulted.Exception
    }
}
```

Task.WhenAny

The WhenAny API is used to asynchronously wait on multiple asynchronous operations represented as Tasks, asynchronously waiting for just one of them to complete. There are four primary uses for WhenAny:

- Redundancy. Doing an operation multiple times and selecting the one that completes first (e.g. contacting multiple stock quote Web services that will all produce a single result and selecting the one that completes the fastest).
- Interleaving. Launching multiple operations and needing them all to complete, but processing them as they complete.
- Throttling. Allowing additional operations to begin as others complete. This is an extension of the interleaving case.
- Early bailout. An operation represented by t1 can be grouped in a WhenAny with another task t2, and we can wait on the WhenAny task. t2 could represent a timeout, or cancellation, or some other signal that will cause the WhenAny task to complete prior to t1 completing.

Redundancy

Consider a case where we want to make a decision about whether to buy a stock. We have several stock recommendation Web services that we trust, but based on daily load each of the services can end

up being fairly slow at different times. We can take advantage of WhenAny to be made aware when any of the operations completes:

```
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol),
    GetBuyRecommendation2Async(symbol),
    GetBuyRecommendation3Async(symbol)
};
Task<bool> recommendation = await Task.WhenAny(recommendations);
if (await recommendation) BuyStock(symbol);
```

Unlike WhenAll, which in the case of successful completion of all tasks returns a list of their unwrapped results, WhenAny returns the Task that completed: if a task fails, it's important to be able to know which failed, and if a task succeeds, it's important to be able to know with which task the returned value is associated. Given this, we need to access the returned task's Result property, or further await it as is done in this example.

As with WhenAll, we need to be able to accommodate exceptions. Due to having received back the completed task, we can await the returned task in order to have errors propagated, and try/catch them appropriately, e.g.

```
Task<bool> [] recommendations = …;
while(recommendations.Count > 0)
{
    Task<bool> recommendation = await Task.WhenAny(recommendations);
    try
    {
        if (await recommendation) BuyStock(symbol);
        break;
    }
    catch(WebException exc)
    {
        recommendations.Remove(recommendation);
    }
}
```

Additionally, even if a first task completes successfully, subsequent tasks may fail. At this point, we have several options in how we deal with their exceptions. One use case may dictate that we not make further forward progress until all of the launched tasks have completed, in which case we may utilize WhenAll. A second use case dictates that all exceptions are important and must be logged. For this, we can utilize continuations directly to receive a notification when tasks have asynchronously completed:

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => { if (t.IsFaulted) Log(t.Exception); });
}
```

or

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => Log(t.Exception), TaskContinuationOptions.OnlyOnFaulted);
}
```

or even:

```
private static async void LogCompletionIfFailed(IEnumerable<Task> tasks)
{
    foreach(var task in tasks)
    {
        try { await task; }
        catch(Exception exc) { Log(exc); }
    }
}
…
LogCompletionIfFailed(recommendations);
```

Finally, the developer may actually want to cancel all of the remaining operations.

```
var cts = new CancellationTokenSource();
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol, cts.Token),
    GetBuyRecommendation2Async(symbol, cts.Token),
    GetBuyRecommendation3Async(symbol, cts.Token)
};

Task<bool> recommendation = await Task.WhenAny(recommendations);
cts.Cancel();
if (await recommendation) BuyStock(symbol);
```

Interleaving

Consider a case where we're downloading images from the Web and doing some processing on each image, such as adding it to a UI control. We need to do the processing sequentially (on the UI thread in the case of the UI control example) but we want to download with as much concurrency as possible, and we don't want to hold up adding the images to the UI until they're all downloaded, but rather add them as they complete:

```
List<Task<Bitmap>> imageTasks =
    (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList();
while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch{}
```

```
}
```

That same interleaving could be applied to a scenario involving not only downloads but also computationally-intensive processing on the ThreadPool of the downloaded images, e.g.

```
List<Task<Bitmap>> imageTasks =
    (from imageUrl in urls select GetBitmapAsync(imageUrl)
         .ContinueWith(t => ConvertImage(t.Result)).ToList();
while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch{}
}
```

Throttling

Consider the same case as in the Interleaving example, except the user is downloading so many images that the downloads need to be explicitly throttled, e.g. only 15 downloads may happen concurrently. To achieve this, a subset of the asynchronous operations may be invoked. As operations complete, additional ones may be invoked to take their place.

```
const int CONCURRENCY_LEVEL = 15;
Uri [] urls = …;
int nextIndex = 0;
var imageTasks = new List<Task<Bitmap>>();
while(nextIndex < CONCURRENCY_LEVEL && nextIndex < urls.Length)
{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}

while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch(Exception exc) { Log(exc); }

    if (nextIndex < urls.Length)
    {
        imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
        nextIndex++;
```

```
    }
}
```

Early Bailout

Consider asynchronously waiting for an operation to complete while simultaneously being responsive to a user's cancellation request (e.g. by clicking a cancel button in the UI).

```
private CancellationTokenSource m_cts;

public void btnCancel_Click(object sender, EventArgs e)
{
    if (m_cts != null) m_cts.Cancel();
}

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokenSource();
    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
        if (imageDownload.IsCompleted)
        {
            Bitmap image = await imageDownload;
            panel.AddImage(image);
        }
        else imageDownload.ContinueWith(t => Log(t));
    }
    finally { btnRun.Enabled = true; }
}

private static async Task UntilCompletionOrCancellation(
    Task asyncOp, CancellationToken ct)
{
    var tcs = new TaskCompletionSource<bool>();
    using(ct.Register(() => tcs.TrySetResult(true)))
        await Task.WhenAny(asyncOp, tcs.Task);
    return asyncOp;
}
```

This implementation reenables the user interface as soon as we decide to bail out but without canceling the underlying asynchronous operations. Another alternative would be to cancel the pending operations when we decide to bail out, however not reestablish the user interface until the operations actually complete, potentially due to ending early due to the cancellation request:

```
private CancellationTokenSource m_cts;

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokenSource();
```

```
        btnRun.Enabled = false;
        try
        {
            Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text, m_cts.Token);
            await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
            Bitmap image = await imageDownload;
            panel.AddImage(image);
        }
        catch(OperationCanceledException) {}
        finally { btnRun.Enabled = true; }
    }
```

Another example of using WhenAny for early bailout involves using Task.WhenAny in conjunction with Task.Delay.

Task.Delay

As shown earlier, the Task.Delay method may be used to introduce pauses into an asynchronous method's execution. This is useful for all kinds of functionality, including building polling loops, delaying the handling of user input for a predetermined period of time, and the like. It can also be useful in combination with Task.WhenAny for implementing timeouts on awaits.

If a task that's part of a larger asynchronous operation (e.g. an ASP.NET Web service) takes too long to complete, the overall operation could suffer, especially if the operation fails to ever complete. Towards this end, it's important to be able to timeout waiting on an asynchronous operation. The synchronous Task.Wait, WaitAll, and WaitAny methods accept timeout values, but the corresponding ContinueWhenAll/Any and the aforementioned WhenAll/WhenAny APIs do not. Instead, Task.Delay and Task.WhenAny may be used in combination to implement a timeout.

Consider a UI application which wants to download an image and disable the UI while the image is downloading. If the download takes too long, however, the UI should be re-enabled and the download should be discarded.

```
public async void btnDownload_Click(object sender, EventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap> download = GetBitmapAsync(url);
        if (download == await Task.WhenAny(download, Task.Delay(3000)))
        {
            Bitmap bmp = await download;
            pictureBox.Image = bmp;
            status.Text = "Downloaded";
        }
        else
        {
            pictureBox.Image = null;
            status.Text = "Timed out";
            var ignored = download.ContinueWith(
                t => Trace("Task finally completed"));
```

```
            }
        }
        finally { btnDownload.Enabled = true; }
    }
```

The same applies to multiple downloads, since WhenAll returns a task:

```
    public async void btnDownload_Click(object sender, RoutedEventArgs e)
    {
        btnDownload.Enabled = false;
        try
        {
            Task<Bitmap[]> downloads =
                Task.WhenAll(from url in urls select GetBitmapAsync(url));
            if (downloads == await Task.WhenAny(downloads, Task.Delay(3000)))
            {
                foreach(var bmp in downloads) panel.AddImage(bmp);
                status.Text = "Downloaded";
            }
            else
            {
                status.Text = "Timed out";
                downloads.ContinueWith(t => Log(t));
            }
        }
        finally { btnDownload.Enabled = true; }
    }
```

Building Task-based Combinators

Due to a task's ability to completely represent an asynchronous operation and provide synchronous and asynchronous capabilities for joining with the operation, retrieving its results, and so forth, it becomes possible to build useful libraries of "combinators" that compose tasks to build larger patterns. As mentioned in the previous section of this document, the .NET Framework includes several built-in combinators, however it's also possible and expected that developers will build their own. Here we provide several examples of potential combinator methods and types.

RetryOnFault

In many situations, it is desirable to retry an operation if a previous attempt at the operation fails. For synchronous code, we might build a helper method to accomplish this as follows:

```
    public static T RetryOnFault<T>(
        Func<T> function, int maxTries)
    {
        for(int i=0; i<maxTries; i++)
        {
            try { return function(); }
            catch { if (i == maxTries-1) throw; }
        }
        return default(T);
    }
```

We can build an almost identical helper method, but for asynchronous operations implemented with the TAP and thus returning tasks:

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

With our function in hand, we can now utilize this combinator to encode retries into our application's logic, e.g.

```
// Download the URL, trying up to three times in case of failure
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3);
```

Our RetryOnFault function could be extended further, such as to accept another Func<Task> which will be invoked between retries in order to determine when it's good to try again, e.g.

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries, Func<Task> retryWhen)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function(); }
        catch { if (i == maxTries-1) throw; }
        await retryWhen().ConfigureAwait(false);
    }
    return default(T);
}
```

which could then be used like the following to wait for a second before retrying:

```
// Download the URL, trying up to three times in case of failure,
// and delaying for a second between retries
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3, () => Task.Delay(1000));
```

NeedOnlyOne

Sometimes redundancy is taken advantage of to improve an operation's latency and chances for success. Consider multiple Web services that all provide stock quotes, but at various times of the day, each of the services may provide different levels of quality and response times. To deal with these, we may issues requests to all of the Web services, and as soon as we get any response, cancel the rest. We can implement a helper function to make easier this common pattern of launching multiple operations, waiting for any, and then canceling the rest:

```
public static async Task<T> NeedOnlyOne(
    params Func<CancellationToken,Task<T>> [] functions)
{
    var cts = new CancellationTokenSource();
    var tasks = (from function in functions
                 select function(cts.Token)).ToArray();
    var completed = await Task.WhenAny(tasks).ConfigureAwait(false);
    cts.Cancel();
    foreach(var task in tasks)
    {
        var ignored = task.ContinueWith(
            t => Log(t), TaskContinuationOptions.OnlyOnFaulted);
    }
    return completed;
}
```

This function can then be used to implement our example:

```
double currentPrice = await NeedOnlyOne(
    ct => GetCurrentPriceFromServer1Async("msft", ct),
    ct => GetCurrentPriceFromServer2Async("msft", ct),
    ct => GetCurrentPriceFromServer3Async("msft", ct));
```

Interleaved

There is a potential performance problem with using Task.WhenAny to support an interleaving scenario when using very large sets of tasks. Every call to WhenAny will result in a continuation being registered with each task, which for N tasks will amount to $O(N^2)$ continuations created over the lifetime of the interleaving operation. To address that if working with a large set of tasks, one could use a combinatory dedicated to the goal:

```
static IEnumerable<Task<T>> Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var sources = (from _ in Enumerable.Range(0, inputTasks.Count)
                   select new TaskCompletionSource<T>()).ToList();
    int nextTaskIndex = -1;
    foreach (var inputTask in inputTasks)
    {
        inputTask.ContinueWith(completed =>
        {
            var source = sources[Interlocked.Increment(ref nextTaskIndex)];
            if (completed.IsFaulted)
                source.TrySetException(completed.Exception.InnerExceptions);
            else if (completed.IsCanceled)
                source.TrySetCanceled();
            else
                source.TrySetResult(completed.Result);
        }, CancellationToken.None,
           TaskContinuationOptions.ExecuteSynchronously,
           TaskScheduler.Default);
    }
    return from source in sources
           select source.Task;
```

```
}
```

This could then be used to process the results of tasks as they complete, e.g.

```
IEnumerable<Task<int>> tasks = ...;
foreach(var task in tasks)
{
    int result = await task;
    …
}
```

WhenAllOrFirstException

In certain scatter/gather scenarios, you might want to wait for all tasks in a set, unless one of them faults, in which case you want to stop waiting as soon as the exception occurs. We can accomplish that with a combinator method as well, for example:

```
public static Task<T[]> WhenAllOrFirstException<T>(IEnumerable<Task<T>> tasks)
{
    var inputs = tasks.ToList();
    var ce = new CountdownEvent(inputs.Count);
    var tcs = new TaskCompletionSource<T[]>();

    Action<Task> onCompleted = (Task completed) =>
    {
        if (completed.IsFaulted)
            tcs.TrySetException(completed.Exception.InnerExceptions);
        if (ce.Signal() && !tcs.Task.IsCompleted)
            tcs.TrySetResult(inputs.Select(t => t.Result).ToArray());
    };

    foreach (var t in inputs) t.ContinueWith(onCompleted);
    return tcs.Task;
}
```

Building Task-based Data Structures

In addition to the ability to build custom task-based combinators, having a data structure in Task and Task<TResult> that represents both the results of an asynchronous operation as well as the necessary synchronization to join with it makes it a very powerful type on which to build custom data structures to be used in asynchronous scenarios.

AsyncCache

One important aspect of Task is that it may be handed out to multiple consumers, all of whom may await it, register continuations with it, get its result (in the case of Task<TResult>) or exceptions, and so on. This makes Task and Task<TResult> perfectly suited to be used in an asynchronous caching infrastructure. Here's a small but powerful asynchronous cache built on top of Task<TResult>:

```
public class AsyncCache<TKey, TValue>
{
    private readonly Func<TKey, Task<TValue>> _valueFactory;
```

```
    private readonly ConcurrentDictionary<TKey, Lazy<Task<TValue>>> _map;

    public AsyncCache(Func<TKey, Task<TValue>> valueFactory)
    {
        if (valueFactory == null) throw new ArgumentNullException("loader");
        _valueFactory = valueFactory;
        _map = new ConcurrentDictionary<TKey, Lazy<Task<TValue>>>();
    }

    public Task<TValue> this[TKey key]
    {
        get
        {
            if (key == null) throw new ArgumentNullException("key");
            return _map.GetOrAdd(key, toAdd =>
                new Lazy<Task<TValue>>(() => _valueFactory(toAdd))).Value;
        }
    }
}
```

The AsyncCache<TKey,TValue> class accepts as a delegate to its constructor a function that takes a TKey and returns a Task<TValue>. Any previously accessed values from the cache are stored in the internal dictionary, with the AsyncCache ensuring that only one task is generated per key, even if the cache is accessed concurrently.

As an example of using this, we could build a cache for downloaded web pages, e.g.

```
private AsyncCache<string,string> m_webPages =
    new AsyncCache<string,string>(DownloadStringAsync);
```

Now, we can use this in asynchronous methods whenever we need the contents of a web page, and the AsyncCache will ensure we're downloading as few pages as possible, caching the results.

```
private async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
    {
        txtContents.Text = await m_webPages["http://www.microsoft.com"];
    }
    finally { btnDownload.IsEnabled = true; }
}
```

AsyncProducerConsumerCollection

Tasks can also be used to build data structures for coordinating between asynchronous activities. Consider one of the classic parallel design patterns: producer/consumer. In producer/consumer, producers generate data which is consumed by consumers, and the producers and consumers may run in parallel (e.g. the consumer processing item 1 which was previously generated by a producer now producing item 2). For producer/consumer, we invariably need some data structure to store the work created by producers so that the consumers may be notified of new data and find it when available.

Here's a simple data structure built on top of tasks that enables asynchronous methods to be used as producers and consumers:

```
public class AsyncProducerConsumerCollection<T>
{
    private readonly Queue<T> m_collection = new Queue<T>();
    private readonly Queue<TaskCompletionSource<T>> m_waiting =
        new Queue<TaskCompletionSource<T>>();

    public void Add(T item)
    {
        TaskCompletionSource<T> tcs = null;
        lock (m_collection)
        {
            if (m_waiting.Count > 0) tcs = m_waiting.Dequeue();
            else m_collection.Enqueue(item);
        }
        if (tcs != null) tcs.TrySetResult(item);
    }

    public Task<T> Take()
    {
        lock (m_collection)
        {
            if (m_collection.Count > 0)
            {
                return Task.FromResult(m_collection.Dequeue());
            }
            else
            {
                var tcs = new TaskCompletionSource<T>();
                m_waiting.Enqueue(tcs);
                return tcs.Task;
            }
        }
    }
}
```

With that in place, we can now write code like the following:

```
private static AsyncProducerConsumerCollection<int> m_data = …;
…
private static async Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.Take();
        ProcessNextItem(nextItem);
    }
}
…
private static void Produce(int data)
{
```

```
        m_data.Add(data);
    }
```

Included in .NET 4.5 is the System.Threading.Tasks.Dataflow.dll assembly. This assembly includes the BufferBlock<T> type, which may be used in a similar manner and without having to build a custom collection type:

```
private static BufferBlock<int> m_data = …;
…
private static async Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.ReceiveAsync();
        ProcessNextItem(nextItem);
    }
}
…
private static void Produce(int data)
{
    m_data.Post(data);
}
```

Interop with Other .NET Asynchronous Patterns and Types

The .NET Framework 1.0 saw the introduction of the IAsyncResult pattern, otherwise known as the Asynchronous Programming Model (APM) pattern, or the Begin/End pattern. The .NET Framework 2.0 then brought with it the event-based asynchronous pattern (EAP). The new TAP deprecates both of its predecessors, while at the same time providing the ability to easily build migration routines from the APM and EAP to TAP.

Tasks and the Asynchronous Programming Model (APM)

From APM to Tasks

The APM pattern relies on two corresponding methods to represent an asynchronous operation: BeginMethodName and EndMethodName. At a high-level, the begin method accepts as parameters to the method the same parameters that would be supplied to the MethodName synchronous method counterpart, as well as also accepting an AsyncCallback delegate and an object state. The begin method then returns an IAsyncResult, which returns from its AsyncState property the object state passed to the begin method. When the asynchronous operation completes, the IAsyncResult's IsCompleted will start returning true, and its AsyncWaitHandle will be set. Additionally, if the AsyncCallback parameter to the begin method was non-null, the callback will be invoked and passed the same IAsyncResult that was returned from the begin method. When the asynchronous operation does complete, the EndMethodName method is used to join with the operation, retrieving any results or forcing any exceptions that occurred to then propagate. There are further details around the IAsyncResult's CompletedSynchronously property that are beyond the scope of this document; for more information, see MSDN.

Given the very structured nature of the APM pattern, it is quite easy to build a wrapper for an APM implementation to expose it as a TAP implementation. In fact, the .NET Framework 4 includes helper routines in the form of TaskFactory.FromAsync to provide this translation.

Consider the .NET Stream class and its BeginRead/EndRead methods, which represent the APM counterpart to the synchronous Read method:

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

Utilizing FromAsync, we can implement a TAP wrapper for this method as follows:

```
public static Task<int> ReadAsync(
    this Stream stream, byte [] buffer, int offset, int count)
{
    if (stream == null) throw new ArgumentNullException("stream");
    return Task<int>.Factory.FromAsync(stream.BeginRead, stream.EndRead,
        buffer, offset, count, null);
}
```

This implementation that utilizes FromAsync is effectively equivalent to the following:

```
public static Task<int> ReadAsync(
    this Stream stream, byte [] buffer, int offset, int count)
{
    if (stream == null) throw new ArgumentNullException("stream");
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, iar =>
    {
        try { tcs.TrySetResult(stream.EndRead(iar)); }
        catch(OperationCanceledException) { tcs.TrySetCanceled(); }
        catch(Exception exc) { tcs.TrySetException(exc); }
    }, null);
    return tcs.Task;
}
```

From Tasks to APM

For cases where existing infrastructure expects code to implement the APM pattern, it is also important to be able to be able to take a TAP implementation and use it where an APM implementation is expected. Thanks to the composability of tasks, and the fact that Task itself implements IAsyncResult, this is achievable with a straightforward helper function (shown here as an extension for Task<TResult>, but an almost identical function may be used for the non-generic Task):

```
public static IAsyncResult AsApm<T>(
    this Task<T> task, AsyncCallback callback, object state)
{
```

```
        if (task == null) throw new ArgumentNullException("task");
        var tcs = new TaskCompletionSource<T>(state);
        task.ContinueWith(t =>
        {
            if (t.IsFaulted) tcs.TrySetException(t.Exception.InnerExceptions)
            else if (t.IsCanceled) tcs.TrySetCanceled();
            else tcs.TrySetResult(t.Result);

            if (callback != null) callback(tcs.Task);
        }, TaskScheduler.Default);
        return tcs.Task;
    }
```

Now, consider a case where we have a TAP implementation:

```
    public static Task<string> DownloadStringAsync(Uri url);
```

and we need to provide an APM implementation:

```
    public IAsyncResult BeginDownloadString(
        Uri url, AsyncCallback callback, object state);
    public string EndDownloadString(IAsyncResult asyncResult);
```

This is achievable with the following code:

```
    public IAsyncResult BeginDownloadString(
        Uri url, AsyncCallback callback, object state)
    {
        return DownloadStringAsync(url).AsApm(callback, state);
    }

    public string EndDownloadString(IAsyncResult asyncResult)
    {
        return ((Task<string>)asyncResult).Result;
    }
```

Tasks and the Event-based Asynchronous Pattern (EAP)

The event-based asynchronous pattern relies on an instance MethodNameAsync method which returns void, accepts the same parameters as the synchronous MethodName method, and initiates the asynchronous operation. Prior to initiating the asynchronous operation, event handlers are registered with events on the same instance, and these events are then raised to provide progress and completion notifications. The event handlers are typically custom delegate types that utilize event argument types that are or that are derived from ProgressChangedEventArgs and AsyncCompletedEventArgs.

Wrapping an EAP implementation is more involved, as the pattern itself involves much more variation and less structure than the APM pattern. To demonstrate, we'll wrap the DownloadStringAsync method. DownloadStringAsync accepts a Uri, raises the DownloadProgressChanged event while downloading in order to report multiple statistics on progress, and raises the DownloadStringCompleted event when done. The final result is a string containing the contents of the page at the specified Uri.

```
    public static Task<string> DownloadStringAsync(Uri url)
```

```
{
    var tcs = new TaskCompletionSource<string>();
    var wc = new WebClient();
    wc.DownloadStringCompleted += (s,e) =>
    {
        if (e.Error != null) tcs.TrySetException(e.Error);
        else if (e.Cancelled) tcs.TrySetCanceled();
        else tcs.TrySetResult(e.Result);
    };
    wc.DownloadStringAsync(url);
    return tcs.Task;
}
```

Tasks and WaitHandles

From WaitHandles to Tasks

While not an asynchronous pattern per-se, advanced developers may find themselves utilizing WaitHandles and the ThreadPool's RegisterWaitForSingleObject method to be notified asynchronously when a WaitHandle is set. We can wrap RegisterWaitForSingleObject to enable a task-based alternative to any synchronous wait on a WaitHandle:

```
public static Task WaitOneAsync(this WaitHandle waitHandle)
{
    if (waitHandle == null) throw new ArgumentNullException("waitHandle");

    var tcs = new TaskCompletionSource<bool>();
    var rwh = ThreadPool.RegisterWaitForSingleObject(waitHandle,
        delegate { tcs.TrySetResult(true); }, null, -1, true);
    var t = tcs.Task;
    t.ContinueWith(_ => rwh.Unregister(null));
    return t;
}
```

With a method like this in hand, we can utilize existing WaitHandle implementations in asynchronous methods. For example, consider the need to throttle the number of asynchronous operations executing at any particular time. For this, we can utilize a System.Threading.Semaphore. By initializing the semaphore's count to N, waiting on the semaphore any time we want to perform an operation, and releasing the semaphore when we're done with an operation, we can throttle to N the number of operations that run concurrently.

```
static Semaphore m_throttle = new Semaphore(N, N);

static async Task DoOperation()
{
    await m_throttle.WaitOneAsync();
    … // do work
    m_throttle.ReleaseOne();
}
```

Using techniques as those demonstrated in this document's previous section on building data structures on top of Task, it is similarly possible to build an asynchronous semaphore that does not rely on WaitHandles and instead works completely in terms of Task. In fact, the SemaphoreSlim type in .NET 4.5 exposes a WaitAsync method that enables this:

For example, the aforementioned BufferBlock<T> type from System.Threading.Tasks.Dataflow.dll may be used towards a similar end:

```
static SemaphoreSlim m_throttle = new SemaphoreSlim(N, N);

static async Task DoOperation()
{
    await m_throttle.WaitAsync();
    … // do work
    m_throttle.Release ();
}
```

From Tasks to WaitHandles

As previously mentioned, the Task class implements IAsyncResult, and its IAsyncResult implementation exposes an AsyncWaitHandle property which returns a WaitHandle that will be set when the Task completes. As such, getting a WaitHandle for a Task is accomplished as follows:

```
WaitHandle wh = ((IAsyncResult)task).AsyncWaitHandle;
```

Case Study: CopyToAsync

The ability to copy one stream to another is a useful and common operation. The Stream.CopyTo instance method was added in .NET 4 to accommodate scenarios that require this functionality, such as downloading the data at a specified URL:

```
public static byte[] DownloadData(string url)
{
    using(var request = WebRequest.Create(url))
    using(var response = request.GetResponse())
    using(var responseStream = response.GetResponseStream())
    using(var result = new MemoryStream())
    {
        responseStream.CopyTo(result);
        return result.ToArray();
    }
}
```

We would like to be able to implement a method like the above with the Task-based Asynchronous Pattern so as to improve responsiveness and scalability. We might attempt to do so as follows:

```
public static async Task<byte[]> DownloadDataAsync(string url)
{
    using(var request = WebRequest.Create(url))
    {
```

```
            return await Task.Run(() =>
            {
                using(var response = request.GetResponse())
                using(var responseStream = response.GetResponseStream())
                using(var result = new MemoryStream())
                {
                    responseStream.CopyTo(result);
                    return result.ToArray();
                }
            }
        }
    }
```

This implementation would improve responsiveness if utilized, for example, from a UI thread, as it offloads from the calling thread the work of downloading the data from the network stream and copying it to the memory stream which will ultimately be used to yield the downloaded data as an array. However, this implementation does not help with scalability, as it's still performing synchronous I/O and blocking a ThreadPool thread in the process while waiting for data to be downloaded. Instead, we would like to be able to write the following function:

```
public static async Task<byte[]> DownloadDataAsync(string url)
{
    using(var request = WebRequest.Create(url))
    using(var response = await request.GetResponseAsync())
    using(var responseStream = response.GetResponseStream())
    using(var result = new MemoryStream())
    {
        await responseStream.CopyToAsync(result);
        return result.ToArray();
    }
}
```

Unfortunately, while Stream has a synchronous CopyTo method, in .NET 4 it lacks an asynchronous CopyToAsync method. We will now walk through providing such an implementation.

A synchronous CopyTo method could be implemented as follows:

```
public static void CopyTo(this Stream source, Stream destination)
{
    var buffer = new byte[0x1000];
    int bytesRead;
    while((bytesRead = source.Read(buffer, 0, buffer.Length)) > 0)
    {
        destination.Write(buffer, 0, bytesRead);
    }
}
```

To provide an asynchronous implementation of CopyTo, utilizing the compiler's ability to implement the TAP, we can modify this implementation slightly:

```
public static async Task CopyToAsync(this Stream source, Stream destination)
{
```

```
    var buffer = new byte[0x1000];
    int bytesRead;
    while((bytesRead = await source.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await destination.WriteAsync(buffer, 0, bytesRead);
    }
}
```

Here, we changed the return type from void to Task, we utilized ReadAsync instead of Read and WriteAsync instead of Write, and we prefixed the calls to ReadAsync and WriteAsync with the await contextual keyword. Following the pattern, we also renamed our method by appending "Async" as a suffix. The ReadAsync and WriteAsync don't exist in .NET 4, but they could be implemented with one statement based on Task.Factory.FromAsync as described in the "Tasks and the Asynchronous Programming Model" section of this docment:

```
public static Task<int> ReadAsync(
    this Stream source, byte [] buffer, int offset, int count)
{
    return Task<int>.Factory.FromAsync(source.BeginRead, source.EndRead,
        buffer, offset, count, null);
}

public static Task WriteAsync(
    this Stream destination, byte [] buffer, int offset, int count)
{
    return Task.Factory.FromAsync(
        destination.BeginWrite, destination.EndWrite,
        buffer, offset, count, null);
}
```

With these methods in hand, we can now successfully implement the CopyToAsync method. We can also optionally support cancellation in the method by adding a CancellationToken that will, for example, be monitored during the copy after every read and write pair (if ReadAsync and/or WriteAsync supported cancellation, the CancellationToken could also be threaded into those calls):

```
public static async Task CopyToAsync(
    this Stream source, Stream destination,
    CancellationToken cancellationToken)
{
    var buffer = new byte[0x1000];
    int bytesRead;
    while((bytesRead = await source.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await destination.WriteAsync(buffer, 0, bytesRead);
        cancellationToken.ThrowIfCancellationRequested();
    }
}
```

(Note that such cancellation could also be useful in a synchronous implementation of CopyTo, and the ability to pass in a CancellationToken enables this. Approaches that would rely on a cancelable object

being returned from the method would receive that object too late, since by the time the synchronous call completed, there would be nothing left to cancel.)

We could also add support for progress notification, including how much data has thus far been copied:

```
public static async Task CopyToAsync(
    this Stream source, Stream destination,
    CancellationToken cancellationToken,
    IProgress<long> progress)
{
    var buffer = new byte[0x1000];
    int bytesRead;
    long totalRead = 0;
    while((bytesRead = await source.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await destination.WriteAsync(buffer, 0, bytesRead);
        cancellationToken.ThrowIfCancellationRequested();
        totalRead += bytesRead;
        progress.Report(totalRead);
    }
}
```

With this method in hand, we can now fully implement our DownloadDataAsync method, including now adding in cancellation and progress support:

```
public static async Task<byte[]> DownloadDataAsync(
    string url,
    CancellationToken cancellationToken,
    IProgress<long> progress)
{
    using(var request = WebRequest.Create(url))
    using(var response = await request.GetResponseAsync())
    using(var responseStream = response.GetResponseStream())
    using(var result = new MemoryStream())
    {
        await responseStream.CopyToAsync(
            result, cancellationToken, progress);
        return result.ToArray();
    }
}
```

Further optimizations are also possible for our CopyToAsync method. For example, if we were to use two buffers instead of one, we could be writing the previously read data while reading in the next piece of data, thus overlapping latencies if both the read and the write are utilizing asynchronous I/O:

```
public static async Task CopyToAsync(this Stream source, Stream destination)
{
    int i = 0;
    var buffers = new [] { new byte[0x1000], new byte[0x1000] };
    Task writeTask = null;
    while(true)
    {
        var readTask = source.ReadAsync(buffers[i], 0, buffers[i].Length))>0;
```

```
            if (writeTask != null) await Task.WhenAll(readTask, writeTask);
            int bytesRead = await readTask;
            if (bytesRead == 0) break;
            writeTask = destination.WriteAsync(buffers[i], 0, bytesRead);
            i ^= 1; // swap buffers
        }
    }
```

Another optimization is to eliminate unnecessary context switches. As mentioned earlier in this document, by default awaiting on a Task will transition back to the SynchronizationContext that was current when the await began. In the case of the CopyToAsync implementation, there's no need to employ such transitions, since we're not manipulating any UI state. We can take advantage of the Task.ConfigureAwait method to disable this automatic switch. For simplicity, changes are shown on the original asynchronous implementation from above:

```
public static Task CopyToAsync(this Stream source, Stream destination)
{
    var buffer = new byte[0x1000];
    int bytesRead;
    while((bytesRead = await
        source.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false)) > 0)
    {
        await destination.WriteAsync(buffer, 0, bytesRead)
            .ConfigureAwait(false);
    }
}
```