Implementing the Task-based Asynchronous Pattern

.NET Framework 4.5

You can implement the Task-based Asynchronous Pattern (TAP) in three ways: by using the C# and Visual Basic compilers in Visual Studio, manually, or through a combination of the compiler and manual methods. The following sections discuss each method in detail. You can use the TAP pattern to implement both compute-bound and I/O-bound asynchronous operations; the Workloads section discusses each type of operation.

Generating TAP Methods

Using the Compilers

In Visual Studio 2012 and the .NET Framework 4.5, any method that is attributed with the **async** keyword (**Async** in Visual Basic) is considered an asynchronous method, and the C# and Visual Basic compilers perform the necessary transformations to implement the method asynchronously by using TAP. An asynchronous method should return either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task<TResult> object. In the case of the latter, the body of the function should return a **TResult**, and the compiler ensures that this result is made available through the resulting task object. Similarly, any exceptions that go unhandled within the body of the method are marshaled to the output task and cause the resulting task to end in the TaskStatus.Faulted state. The exception is when an OperationCanceledException (or derived type) goes unhandled, in which case the resulting task ends in the TaskStatus.Canceled state.

Generating TAP Methods Manually

You may implement the TAP pattern manually for better control over implementation. The compiler relies on the public surface area exposed from the System.Threading.Tasks namespace and supporting types in the System.Runtime.CompilerServices namespace. To implement the TAP yourself, you create a TaskCompletionSource<TResult> object, perform the asynchronous operation, and when it completes, call the SetResult, SetException, or SetCanceled method, or the **Try** version of one of these methods. When you implement a TAP method manually, you must complete the resulting task when the represented asynchronous operation completes. For example:

C#

```
public static Task<int> ReadTask(this Stream stream, byte[] buffer, int offset, int count, object state)
{
    var tcs = new TaskCompletionSource<int>();
    stream.BeginRead(buffer, offset, count, ar =>
    {
        try { tcs.SetResult(stream.EndRead(ar)); }
        catch (Exception exc) { tcs.SetException(exc); }
    }, state);
    return tcs.Task;
}
```

Hybrid Approach

You may find it useful to implement the TAP pattern manually but to delegate the core logic for the implementation to the compiler. For example, you may want to use the hybrid approach when you want to verify arguments outside a compiler-generated asynchronous method so that exceptions can escape to the method's direct caller rather than being exposed through the System.Threading.Tasks.Task object:

C#

```
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 goes here

   return value;
}
```

Another case where such delegation is useful is when you're implementing fast-path optimization and want to return a cached task.

Workloads

You may implement both compute-bound and I/O-bound asynchronous operations as TAP methods. However, when TAP methods are exposed publicly from a library, they should be provided only for workloads that involve I/O-bound operations (they may also involve computation, but should not be purely computational). If a method is purely compute-bound, it should be exposed only as a synchronous implementation; the code that consumes it may then choose whether to wrap an invocation of that synchronous method into a task to offload the work to another thread or to achieve parallelism.

Compute-bound Tasks

The System.Threading.Tasks.Task class is ideally suited for representing computationally intensive operations. By default, it takes advantage of special support within the ThreadPool class to provide efficient execution, and it also provides significant control over when, where, and how asynchronous computations execute.

You can generate compute-bound tasks in the following ways:

- In the .NET Framework 4, use the TaskFactory.StartNew method, which accepts a delegate (typically an Action<T> or a Func<TResult>) to be executed asynchronously. If you provide an Action<T> delegate, the method returns a System.Threading.Tasks.Task object that represents the asynchronous execution of that delegate. If you provide a Func<TResult> delegate, the method returns a System.Threading.Tasks.Task<TResult> object. Overloads of the StartNew method accept a cancellation token (CancellationToken), task creation options (TaskCreationOptions), and a task scheduler (TaskScheduler), all of which provide fine-grained control over the scheduling and execution of the task. A factory instance that targets the current task scheduler is available as a static property (Factory) of the Task class; for example: `Task.Factory.StartNew(…)`.

- In the .NET Framework 4.5, use the static Task.Run method as a shortcut to TaskFactory.StartNew. You may use Run to easily launch a compute-bound task that targets the thread pool. In the .NET Framework 4.5, this is the preferred mechanism for launching a compute-bound task. Use **StartNew** directly only when you want more fine-grained control over the task.
- Use the constructors of the **Task** type or the **Start** method if you want to generate and schedule the task separately. Public methods must only return tasks that have already been started.
- Use the overloads of the Task.ContinueWith method. This method creates a new task that is scheduled when another task completes. Some of the ContinueWith overloads accept a cancellation token, continuation options, and a task scheduler for better control over the scheduling and execution of the continuation task.
- Use the TaskFactory.ContinueWhenAll and TaskFactory.ContinueWhenAny methods. These methods create a new task that is scheduled when all or any of a supplied set of tasks completes. These methods also provide overloads to control the scheduling and execution of these tasks.

In compute-bound tasks, the system can prevent the execution of a scheduled task if it receives a cancellation request before it starts running the task. As such, if you provide a cancellation token (CancellationToken object), you can pass that token to the asynchronous code that monitors the token. You can also provide the token to one of the previously mentioned methods such as **StartNew** or **Run** so that the **Task** runtime may also monitor the token.

For example, consider an asynchronous method that renders an image. The body of the task can poll the cancellation token so that the code may exit early if a cancellation request arrives during rendering. In addition, if the cancellation request arrives before rendering starts, you'll want to prevent the rendering operation:

C#

```
internal 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 end in a Canceled state if at least one of the following conditions is true:

- A cancellation request arrives through the CancellationToken object, which is provided as an argument to the creation method (for example, **StartNew** or **Run**) before the task transitions to the Running state.
- An OperationCanceledException exception goes unhandled within the body of such a task, that exception contains the same CancellationToken that is passed to the task, and that token shows that cancellation is requested.

If another exception goes unhandled within the body of the task, the task ends in the Faulted state, and any attempts to wait on the task or access its result causes an exception to be thrown.

I/O-bound Tasks

To create a task that should not be directly backed by a thread for the entirety of its execution, use the TaskCompletionSource<TResult> type. This type exposes a Task property that returns an associated Task<TResult> instance. The life cycle of this task is controlled by TaskCompletionSource<TResult> methods such as SetResult, SetException, SetCanceled, and their **TrySet** variants.

Let's say that you want to create a task that will complete after a specified period of time. For example, you may want to delay an activity in the user interface. The System.Threading.Timer class already provides the ability to asynchronously invoke a delegate after a specified period of time, and by using TaskCompletionSource<TResult> you can put a Task<TResult> front on the timer, for example:

C#

```
public static Task<DateTimeOffset> Delay(int millisecondsTimeout)
{
    TaskCompletionSource<DateTimeOffset> tcs = null;
    Timer timer = null;

    timer = new Timer(delegate
    {
        timer.Dispose();
        tcs.TrySetResult(DateTimeOffset.UtcNow);
    }, null, Timeout.Infinite, Timeout.Infinite);

    tcs = new TaskCompletionSource<DateTimeOffset>(timer);
    timer.Change(millisecondsTimeout, Timeout.Infinite);
    return tcs.Task;
}
```

Starting with the .NET Framework 4.5, the Task.Delay method is provided for this purpose, and you can use it inside another asynchronous method, for example, to implement an asynchronous polling loop:

C#

```
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);
    }
```

```
}
```

The TaskCompletionSource<TResult> class doesn't have a non-generic counterpart. However, Task<TResult> derives from Task, so you can use the generic TaskCompletionSource<TResult> object for I/O-bound methods that simply return a task. To do this, you can use a source with a dummy **TResult** (Boolean is a good default choice, but if you're concerned about the user of the Task downcasting it to a Task<TResult>, you can use a private **TResult** type instead). For example, the `Delay` method in the previous example returns the current time along with the resulting offset (`Task<DateTimeOffset>`). If such a result value is unnecessary, the method could instead be coded as follows (note the change of return type and the change of argument to TrySetResult):

C#

```
public static Task<bool> Delay(int millisecondsTimeout)
{
     TaskCompletionSource<bool> tcs = null;
     Timer timer = null;

     timer = new Timer(delegate
     {
         timer.Dispose();
         tcs.TrySetResult(true);
     }, null, Timeout.Infinite, Timeout.Infinite);

     tcs = new TaskCompletionSource<bool>(timer);
     timer.Change(millisecondsTimeout, Timeout.Infinite);
     return tcs.Task;
}
```

Mixed Compute-bound 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, multiple asynchronous operations are often combined into larger mixed operations. For example, the `RenderAsync` method in a previous example performed a computationally intensive operation to render an image based on some input `imageData`. This `imageData` could come from a web service that you asynchronously access:

C#

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

This example also demonstrates how a single cancellation token may be threaded through multiple asynchronous operations. For more information, see the cancellation usage section in Consuming the Task-based Asynchronous Pattern.

See Also

Concepts
Consuming the Task-based Asynchronous Pattern
Interop with Other Asynchronous Patterns and Types
Other Resources
Task-based Asynchronous Pattern (TAP)