Table of Contents

Asynchronous Programming Patterns

11/11/2017 • 2 min to read • Edit Online

The .NET Framework provides three patterns for performing asynchronous operations:

- Asynchronous Programming Model (APM) pattern (also called the IAsyncResult pattern), where asynchronous operations require `Begin` and `End` methods (for example, `BeginWrite` and `EndWrite` for asynchronous write operations). This pattern is no longer recommended for new development. For more information, see Asynchronous Programming Model (APM).
- Event-based Asynchronous Pattern (EAP), which requires a method that has the `Async` suffix, and also requires one or more events, event handler delegate types, and `EventArg`-derived types. EAP was introduced in the .NET Framework 2.0. It is no longer recommended for new development. For more information, see Event-based Asynchronous Pattern (EAP).
- Task-based Asynchronous Pattern (TAP), which uses a single method to represent the initiation and completion of an asynchronous operation. TAP was introduced in the .NET Framework 4 and is the recommended approach to asynchronous programming in the .NET Framework. The async and await keywords in C# and the Async and Await operators in Visual Basic Language add language support for TAP. For more information, see Task-based Asynchronous Pattern (TAP).

Comparing Patterns

For a quick comparison of how the three patterns model asynchronous operations, 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 of this method would expose the `BeginRead` and `EndRead` 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;
}
```

The TAP counterpart would expose the following single `ReadAsync` method:

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

For a comprehensive discussion of TAP, APM, and EAP, see the links provided in the next section.

Related topics

TITLE	DESCRIPTION
Asynchronous Programming Model (APM)	Describes the legacy model that uses the IAsyncResult interface to provide asynchronous behavior. This model is no longer recommended for new development.
Event-based Asynchronous Pattern (EAP)	Describes the event-based legacy model for providing asynchronous behavior. This model is no longer recommended for new development.
Task-based Asynchronous Pattern (TAP)	Describes the new asynchronous pattern based on the System.Threading.Tasks namespace. This model is the recommended approach to asynchronous programming in the .NET Framework 4 and later versions.

See also

Asynchronous programming in C#
Async Programming in F#
Asynchronous Programming with Async and Await (Visual Basic)

Task-based Asynchronous Pattern (TAP)

11/11/2017 • 12 min to read • Edit Online

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

Naming, Parameters, and Return Types

TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the Asynchronous Programming Model (APM or `IAsyncResult`) pattern, which requires `Begin` and `End` methods, and in contrast to the Event-based Asynchronous Pattern (EAP), which requires a method that has the `Async` suffix and also requires one or more events, event handler delegate types, and `EventArg`-derived types. Asynchronous methods in TAP include the `Async` suffix after the operation name; for example, `GetAsync` for a `Get` operation. If you're adding a TAP method to a class that already contains that method name with the `Async` suffix, use the suffix `TaskAsync` instead. For example, if the class already has a `GetAsync` method, use the name `GetTaskAsync`.

The TAP method returns either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task<TResult>, based on whether the corresponding synchronous method returns void or a type `TResult`.

The parameters of a TAP method should match the parameters of its synchronous counterpart, and should be provided in the same order. However, `out` and `ref` parameters are exempt 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 `TResult` returned by Task<TResult>, and should use a tuple or a custom data structure to accommodate multiple values. Methods that are devoted exclusively 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 to which the method belongs) need not follow this naming pattern; such methods are often referred to as *combinators*. Examples of combinators include WhenAll and WhenAny, and are discussed in the Using the Built-in Task-based Combinators section of the article Consuming the Task-based Asynchronous Pattern.

For examples of how the TAP syntax differs from the syntax used in legacy asynchronous programming patterns such as the Asynchronous Programming Model (APM) and the Event-based Asynchronous Pattern (EAP), see Asynchronous Programming Patterns.

Initiating an Asynchronous Operation

An asynchronous method that is based on TAP can do a small amount of work synchronously, such as validating arguments and initiating the asynchronous operation, before it returns the resulting task. Synchronous work should be kept to the minimum so the asynchronous method can return quickly. Reasons for a quick return include the following:

- Asynchronous methods may be invoked from user interface (UI) threads, and any long-running synchronous work could harm the responsiveness of the application.
- Multiple asynchronous methods may be launched concurrently. Therefore, any long-running work in the synchronous portion of an asynchronous method could delay the initiation of other asynchronous operations, thereby decreasing the benefits of concurrency.

In some cases, the amount of work required to complete the operation is less than the amount of work required to

launch the operation asynchronously. Reading from a stream where the read operation can be satisfied by data that is already buffered in memory is an example of such a scenario. In such cases, the operation may complete synchronously, and may return a task that has already been completed.

Exceptions

An asynchronous method should raise an exception to be thrown out of the asynchronous method call only in response to a usage error. Usage errors should never occur in production code. For example, if passing a null reference (`Nothing` in Visual Basic) as one of the method's arguments causes an error state (usually represented by an ArgumentNullException exception), you can modify the calling code to ensure that a null reference is never passed. For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned. Typically, a task contains at most one exception. However, if the task represents multiple operations (for example, WhenAll), multiple exceptions may be associated with a single task.

Target Environment

When you implement a TAP method, you can determine where asynchronous execution occurs. You may choose to execute the workload on the thread pool, implement it by using asynchronous I/O (without being bound to a thread for the majority of the operation's execution), run it on a specific thread (such as the UI thread), or use any number of potential contexts. A TAP method may even have nothing to execute, and may just return a Task that represents the occurrence of a condition elsewhere in the system (for example, a task that represents data 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 run additional (continuation) code when the asynchronous operation completes. The creator of the continuation code has control over where that code executes. You may create the continuation code either explicitly, through methods on the Task class (for example, ContinueWith) or implicitly, by using language support built on top of continuations (for example, `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. To support corner cases of types that derive from Task and Task<TResult>, and to support the separation of construction from scheduling, the Task class exposes a Start method. Tasks that are created by the public Task constructors are referred to as *cold tasks*, because they begin their life cycle in the non-scheduled Created state and are scheduled only when Start is called on these instances.

All other tasks begin their life cycle in a hot state, which means that the asynchronous operations they represent have already been initiated and their task status is an enumeration value other than TaskStatus.Created. All tasks that are returned from TAP methods must be activated. **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 before returning it.** Consumers of a TAP method may safely assume that the returned task is active and should not try to call Start on any Task that is returned from a TAP method. Calling Start on an active task results in an InvalidOperationException exception.

Cancellation (Optional)

In TAP, cancellation is optional for both asynchronous method implementers and asynchronous method consumers. If an operation allows cancellation, it exposes an overload of the asynchronous method that accepts a cancellation token (CancellationToken instance). By convention, the parameter is named `cancellationToken`.

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

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          cancellationToken As CancellationToken) _
                          As Task
```

The asynchronous operation monitors this token for cancellation requests. If it receives a cancellation request, it may choose to honor that request and cancel the operation. If the cancellation request results in work being ended prematurely, the TAP method returns a task that ends in the Canceled state; there is no available result and no exception is thrown. The Canceled state is considered to be a final (completed) state for a task, along with the Faulted and RanToCompletion states. Therefore, if a task is in the Canceled state, its IsCompleted property returns `true`. When a task completes in the Canceled state, any continuations registered with the task are scheduled or executed, unless a continuation option such as NotOnCanceled was specified to opt out of continuation. Any code that is asynchronously waiting for a canceled task through use of language features continues to run but receives an OperationCanceledException or an exception derived from it. Code that is blocked synchronously waiting on the task through methods such as Wait and WaitAll also continue to run with an exception.

If a cancellation token has requested cancellation before the TAP method that accepts that token is called, the TAP method should return a Canceled task. However, if cancellation is requested while the asynchronous operation is running, the asynchronous operation need not accept the cancellation request. The returned task should end in the Canceled state only if the operation ends as a result of the cancellation request. If cancellation is requested but a result or an exception is still produced, the task should end in the RanToCompletion or Faulted state. For asynchronous methods used by a developer who wants cancellation first and foremost, you don't have to provide an overload that doesn't accept a cancellation token. For methods that cannot be canceled, do not provide overloads that accept a cancellation token; this helps indicate to the caller whether the target method is actually cancelable. Consumer code that does not desire cancellation may call a method that accepts a CancellationToken and provide None as the argument value. None is functionally equivalent to the default CancellationToken.

Progress Reporting (Optional)

Some asynchronous operations benefit from providing progress notifications; these are typically used to update a user interface with information about the progress of the asynchronous operation. In TAP, progress is handled through an IProgress<T> interface, which is passed to the asynchronous method as a parameter that is usually named `progress`. Providing the progress interface when the asynchronous method is called helps eliminate race conditions that result from incorrect usage (that is, when event handlers that are incorrectly registered after the operation starts may miss updates). More importantly, the progress interface supports varying implementations of progress, as determined by the consuming code. For example, the consuming code may only care about the latest progress update, or may want to buffer all updates, or may want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread. All these options may be achieved by using a different implementation of the interface, customized to the particular consumer's needs. As with cancellation, TAP implementations should provide an IProgress<T> parameter only if the API supports progress notifications. For example, if the `ReadAsync` method discussed earlier in this article is able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an IProgress<T> interface:

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

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          progress As IProgress(Of Long)) As Task
```

If a `FindFilesAsync` method returns a list of all files that meet a particular search pattern, the progress callback could provide an estimate of the percentage of work completed as well as the current set of partial results. It could do this either with a tuple:

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

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of Tuple(Of Double, ReadOnlyCollection(Of List(Of
FileInfo))))) _
                               As  Task(Of ReadOnlyCollection(Of FileInfo))
```

or with a data type that is specific to the API:

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

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of FindFilesProgressInfo)) _
                               As Task(Of ReadOnlyCollection(Of FileInfo))
```

In the latter case, the special data type is usually 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 report the progress to the Progress<T> object synchronously, which enables the asynchronous method to quickly provide progress, and allow the consumer of the progress to determine how and where best to handle the information. For example, the progress instance could choose to marshal callbacks and raise events on a captured synchronization context.

IProgress<T> Implementations

The .NET Framework 4.5 provides a single IProgress<T> implementation: Progress<T>. 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;
}
```

```
Public Class Progress(Of T) : Inherits IProgress(Of T)
    Public Sub New()
    Public Sub New(handler As Action(Of T))
    Protected Overridable Sub OnReport(value As T)
    Public Event ProgressChanged As EventHandler(Of T>
End Class
```

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 the SynchronizationContext object that was captured when the Progress<T> instance was instantiated. If no synchronization context was available, a default context that targets the thread pool is used. Handlers may be registered with this event. A single handler may also be provided to the Progress<T> constructor for convenience, and behaves just like an event handler for the ProgressChanged event. Progress updates are raised asynchronously to avoid delaying the asynchronous operation while event handlers are executing. Another IProgress<T> implementation could choose to apply different semantics.

Choosing the Overloads to Provide

If a TAP implementation uses both the optional CancellationToken and optional IProgress<T> parameters, it could potentially require 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);
```

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken cancellationToken) As Task
Public MethodNameAsync(…, progress As IProgress(Of T)) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken,
                       progress As IProgress(Of T)) As Task
```

However, many TAP implementations provide neither cancellation or progress capabilities, so they require a single method:

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

```
Public MethodNameAsync(…) As Task
```

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

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

// … or …

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

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken) As Task

' … or …

Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, progress As IProgress(Of T)) As Task
```

If a TAP implementation supports both cancellation and progress, it may expose all four overloads. However, it may provide only the following two:

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

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken,
                       progress As IProgress(Of T)) As Task
```

To compensate for the two missing intermediate combinations, developers may pass None or a default CancellationToken for the `cancellationToken` parameter and `null` for the `progress` parameter.

If you expect every usage of the TAP method to support cancellation or progress, you may omit the overloads that don't accept the relevant parameter.

If you decide to expose multiple overloads to make cancellation or progress optional, the overloads that don't support cancellation or progress should behave as if they passed None for cancellation or `null` for progress to the overload that does support these.

Related Topics

TITLE	DESCRIPTION
Asynchronous Programming Patterns	Introduces the three patterns for performing asynchronous operations: the Task-based Asynchronous Pattern (TAP), the Asynchronous Programming Model (APM), and the Event-based Asynchronous Pattern (EAP).
Implementing the Task-based Asynchronous Pattern	Describes how to 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.
Consuming the Task-based Asynchronous Pattern	Describes how you can use tasks and callbacks to achieve waiting without blocking.
Interop with Other Asynchronous Patterns and Types	Describes how to use the Task-based Asynchronous Pattern (TAP) to implement the Asynchronous Programming Model (APM) and Event-based Asynchronous Pattern (EAP).

Implementing the Task-based Asynchronous Pattern

11/11/2017 • 9 min to read • Edit Online

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

Starting with .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. For 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:

```
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;
}
```

```
<Extension()>
Public Function ReadTask(stream As Stream, buffer() As Byte,
                         offset As Integer, count As Integer,
                         state As Object) As Task(Of Integer)
    Dim tcs As New TaskCompletionSource(Of Integer)()
    stream.BeginRead(buffer, offset, count, Sub(ar)
               Try
                  tcs.SetResult(stream.EndRead(ar))
               Catch exc As Exception
                  tcs.SetException(exc)
               End Try
            End Sub, state)
    Return tcs.Task
End Function
```

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:

```
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;
}
```

```
Public Function MethodAsync(input As String) As Task(Of Integer)
    If input Is Nothing Then Throw New ArgumentNullException("input")

    Return MethodAsyncInternal(input)
End Function

Private Async Function MethodAsyncInternal(input As String) As Task(Of Integer)

   ' code that uses await goes here

   return value
End Function
```

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. And if a method is IO-bound, it should be exposed only as an asynchronous implementation.

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 and later versions (including .NET Core and .NET Standard), 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 and later versions, 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:

```
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);
}
```

```
Friend Function RenderAsync(data As ImageData, cancellationToken As _
                            CancellationToken) As Task(Of Bitmap)
    Return Task.Run( Function()
                       Dim bmp As New Bitmap(data.Width, data.Height)
                       For y As Integer = 0 to data.Height - 1
                          cancellationToken.ThrowIfCancellationRequested()
                          For x As Integer = 0 To data.Width - 1
                            ' render pixel [x,y] into bmp
                          Next
                       Next
                       Return bmp
                     End Function, cancellationToken)
End Function
```

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:

```
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;
}
```

```
Public Function Delay(millisecondsTimeout As Integer) As Task(Of DateTimeOffset)
    Dim tcs As TaskCompletionSource(Of DateTimeOffset) = Nothing
    Dim timer As Timer = Nothing

    timer = New Timer( Sub(obj)
                          timer.Dispose()
                          tcs.TrySetResult(DateTimeOffset.UtcNow)
                       End Sub, Nothing, Timeout.Infinite, Timeout.Infinite)

    tcs = New TaskCompletionSource(Of DateTimeOffset)(timer)
    timer.Change(millisecondsTimeout, Timeout.Infinite)
    Return tcs.Task
End Function
```

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:

```
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);
    }
}
```

```
Public Async Function Poll(url As Uri, cancellationToken As CancellationToken,
                           progress As IProgress(Of Boolean)) As Task
    Do While True
        Await Task.Delay(TimeSpan.FromSeconds(10), cancellationToken)
        Dim success As Boolean = False
        Try
            await DownloadStringAsync(url)
            success = true
        Catch
           ' ignore errors
        End Try
        progress.Report(success)
    Loop
End Function
```

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):

```
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;
}
```

```
Public Function Delay(millisecondsTimeout As Integer) As Task(Of Boolean)
     Dim tcs As TaskCompletionSource(Of Boolean) = Nothing
     Dim timer As Timer = Nothing

     Timer = new Timer( Sub(obj)
                           timer.Dispose()
                           tcs.TrySetResult(True)
                       End Sub, Nothing, Timeout.Infinite, Timeout.Infinite)

     tcs = New TaskCompletionSource(Of Boolean)(timer)
     timer.Change(millisecondsTimeout, Timeout.Infinite)
     Return tcs.Task
End Function
```

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:

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

```
Public Async Function DownloadDataAndRenderImageAsync(
             cancellationToken As CancellationToken) As Task(Of Bitmap)
    Dim imageData As ImageData = Await DownloadImageDataAsync(cancellationToken)
    Return Await RenderAsync(imageData, cancellationToken)
End Function
```

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

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

Consuming the Task-based Asynchronous Pattern

11/21/2017 • 22 min to read • Edit Online

When you use the Task-based Asynchronous Pattern (TAP) to work with asynchronous operations, you can use callbacks to achieve waiting without blocking. For tasks, this is achieved through methods such as Task.ContinueWith. Language-based asynchronous support hides callbacks by allowing asynchronous operations to be awaited within normal control flow, and compiler-generated code provides this same API-level support.

Suspending Execution with Await

Starting with the .NET Framework 4.5, you can use the await keyword in C# and the Await Operator in Visual Basic to asynchronously await Task and Task<TResult> objects. When you're awaiting a Task, the `await` expression is of type `void`. When you're awaiting a Task<TResult>, the `await` expression is of type `TResult`. An `await` expression must occur inside the body of an asynchronous method. For more information about 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 by using a continuation. This callback resumes 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` is returned. If the Task or Task<TResult> that was awaited ended in the Canceled state, an OperationCanceledException exception is thrown. If the Task or Task<TResult> that was awaited ended in the Faulted state, the exception that caused it to fault is thrown. A `Task` can fault as a result of multiple exceptions, but only one of these exceptions is propagated. However, the Task.Exception property returns an AggregateException exception that contains all the errors.

If a synchronization context (SynchronizationContext object) is associated with the thread that was executing the asynchronous method at the time of suspension (for example, if the SynchronizationContext.Current property is not `null`), the asynchronous method resumes on that same synchronization context by using the context's Post method. Otherwise, it relies on the task scheduler (TaskScheduler object) that was current at the time of suspension. Typically, this is the default task scheduler (TaskScheduler.Default), which targets the thread pool. This task scheduler determines whether the awaited asynchronous operation should resume where it completed or whether the resumption should be scheduled. The default scheduler typically allows the continuation to run on the thread that the awaited operation completed.

When an asynchronous method is called, it synchronously executes the body of the function up until the first await expression on an awaitable instance that has 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> object 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 is eventually published.

There are several important variations of this behavior. For performance reasons, if a task has already completed by the time the task is awaited, control is not yielded, and the function continues to execute. Additionally, returning to the original context isn't always the desired behavior and can be changed; this is described in more detail in the next section.

Configuring Suspension and Resumption with Yield and ConfigureAwait

Several methods provide more control over an asynchronous method's execution. For example, you can use the Task.Yield method 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 the current context.

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

You can also use the Task.ConfigureAwait method for better control over suspension and resumption in an asynchronous method. As mentioned previously, by default, the current context is captured at the time an asynchronous method is suspended, and that captured context is used to invoke the asynchronous method's continuation upon resumption. In many cases, this is the exact behavior you want. In other cases, you may not care about the continuation context, and you can achieve better performance by avoiding such posts back to the original context. To enable this, use the Task.ConfigureAwait method to inform the await operation not to capture and resume on the context, but to continue execution wherever the asynchronous operation that was being awaited completed:

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

Canceling an Asynchronous Operation

Starting with the .NET Framework 4, TAP methods that support cancellation provide at least one overload that accepts a cancellation token (CancellationToken object).

A cancellation token is created through a cancellation token source (CancellationTokenSource object). The source's Token property returns the cancellation token that will be signaled when the source's Cancel method is called. For example, if you want to download a single webpage and you want to be able to cancel the operation, you create a CancellationTokenSource object, pass its token to the TAP method, and then call the source's Cancel method when you're ready to cancel the operation:

```
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, you can pass the same token to all invocations:

```
var cts = new CancellationTokenSource();
    IList<string> results = await Task.WhenAll(from url in urls select DownloadStringAsync(url, cts.Token));
    // at some point later, potentially on another thread
    …
    cts.Cancel();
```

Or, you can pass the same token to a selective subset of operations:

```
var cts = new CancellationTokenSource();
    byte [] data = await DownloadDataAsync(url, cts.Token);
    await SaveToDiskAsync(outputPath, data, CancellationToken.None);
    … // at some point later, potentially on another thread
    cts.Cancel();
```

Cancellation requests may be initiated from any thread.

You can pass the CancellationToken.None value to any method that accepts a cancellation token to indicate that cancellation will never be requested. This causes the CancellationToken.CanBeCanceled property to return `false`, and the called method can optimize accordingly. For testing purposes, you can also pass in a pre-canceled cancellation token that is instantiated by using the constructor that accepts a Boolean value to indicate whether the token should start in an already-canceled or not-cancelable state.

This approach to cancellation has several advantages:

- You can pass the same cancellation token to any number of asynchronous and synchronous operations.
- The same cancellation request may be proliferated to any number of listeners.
- The developer of the asynchronous API is in complete control of whether cancellation may be requested and when it may take effect.
- The code that consumes the API may selectively determine the asynchronous invocations that cancellation requests will be propagated to.

Monitoring 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 (WPF) 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 methods for composing and working with tasks.

Task.Run

The Task class includes several Run methods that let you easily offload work as a Task or Task<TResult> to the thread pool, for example:

```
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 Task.Run(Func<Task>) overload, exist as shorthand for the TaskFactory.StartNew method. Other overloads, such as Task.Run(Func<Task>), enable you to use await within the offloaded work, for example:

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

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

Task.FromResult

Use the FromResult method in scenarios where data may already be available and just needs to be returned from a task-returning method lifted into a Task<TResult>:

```
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

Use the WhenAll method to asynchronously wait on multiple asynchronous operations that are represented as tasks. The method has multiple overloads that support a set of non-generic tasks or a non-uniform set of generic tasks (for example, asynchronously waiting for multiple void-returning operations, or asynchronously waiting for multiple value-returning methods where each value may have a different type) and to support a uniform set of generic tasks (such as asynchronously waiting for multiple `TResult`-returning methods).

Let's say you want to send email messages to several customers. You can overlap sending the messages so you're not waiting for one message to complete before sending the next. You can also find out when the send operations have completed and whether any errors have occurred:

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

This code doesn't explicitly handle exceptions that may occur, but lets exceptions propagate out of the `await` on the resulting task from WhenAll. To handle the exceptions, you can use code such as the following:

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

In this case, if any asynchronous operation fails, all the exceptions will be consolidated in an AggregateException exception, which is stored in the Task that is returned from the WhenAll method. However, only one of those exceptions is propagated by the `await` keyword. If you want to examine all the exceptions, you can rewrite the previous code 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
    }
}
```

Let's consider an example of downloading multiple files from the web asynchronously. In this case, all the asynchronous operations have homogeneous result types, and it's easy to access the results:

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

You can use the same exception-handling techniques we discussed in the previous void-returning scenario:

```
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

You can use the WhenAny method to asynchronously wait for just one of multiple asynchronous operations represented as tasks to complete. This method serves four primary use cases:

- Redundancy: Performing an operation multiple times and selecting the one that completes first (for example, contacting multiple stock quote web services that will produce a single result and selecting the one that completes the fastest).
- Interleaving: Launching multiple operations and waiting for all of them to complete, but processing them as they complete.
- Throttling: Allowing additional operations to begin as others complete. This is an extension of the interleaving scenario.
- Early bailout: For example, an operation represented by task t1 can be grouped in a WhenAny task with another task t2, and you can wait on the WhenAny task. Task t2 could represent a time-out, or cancellation, or some other signal that causes the WhenAny task to complete before t1 completes.

Redundancy

Consider a case where you want to make a decision about whether to buy a stock. There are several stock recommendation web services that you trust, but depending on daily load, each service can end up being slow at different times. You can use the WhenAny method to receive a notification when any operation 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 returns the unwrapped results of all tasks that completed successfully, WhenAny returns the task that completed. If a task fails, it's important to know that it failed, and if a task succeeds, it's important to know which task the return value is associated with. Therefore, you need to access the result of the returned task, or further await it, as this example shows.

As with WhenAll, you have to be able to accommodate exceptions. Because you receive the completed task back, you can await the returned task to have errors propagated, and `try/catch` them appropriately; for example:

```
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, you have several options for dealing with exceptions: You can wait until all the launched tasks have completed, in which case you can use the WhenAll method, or you can decide that all exceptions are important and must be logged. For this, you can use continuations to receive a notification when tasks have completed asynchronously:

```
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, you may want to cancel all 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 you're downloading images from the web and processing each image (for example, adding the image to a UI control). You have to do the processing sequentially on the UI thread, but you want to download the images as concurrently as possible. Also, you don't want to hold up adding the images to the UI until they're all downloaded—you want to 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{}
}
```

You can also apply interleaving to a scenario that involves computationally intensive processing on the ThreadPool of the downloaded images; for example:

```
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 interleaving example, except that the user is downloading so many images that the downloads have to be throttled; for example, you want only a specific number of downloads to happen concurrently. To achieve this, you can start a subset of the asynchronous operations. As operations complete, you can start additional operations 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 that you're waiting asynchronously for an operation to complete while simultaneously responding to a user's cancellation request (for example, the user clicked a cancel button). The following code illustrates this scenario:

```
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 re-enables the user interface as soon as you decide to bail out, but doesn't cancel the underlying asynchronous operations. Another alternative would be to cancel the pending operations when you decide to bail out, but 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 early bailout involves using the WhenAny method in conjunction with the Delay method, as discussed in the next section.

Task.Delay

You can use the Task.Delay method to introduce pauses into an asynchronous method's execution. This is useful

for many kinds of functionality, including building polling loops and delaying the handling of user input for a predetermined period of time. The Task.Delay method can also be useful in combination with Task.WhenAny for implementing time-outs on awaits.

If a task that's part of a larger asynchronous operation (for example, an ASP.NET web service) takes too long to complete, the overall operation could suffer, especially if it fails to ever complete. For this reason, it's important to be able to time out when waiting on an asynchronous operation. The synchronous Task.Wait, Task.WaitAll, and Task.WaitAny methods accept time-out values, but the corresponding TaskFactory.ContinueWhenAll/Task.WhenAny and the previously mentioned Task.WhenAll/Task.WhenAny methods do not. Instead, you can use Task.Delay and Task.WhenAny in combination to implement a time-out.

For example, in your UI application, let's say that you want to download an image and disable the UI while the image is downloading. However, if the download takes too long, you want to re-enable the UI and discard the download:

```
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, because 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

Because a task is able to completely represent an asynchronous operation and provide synchronous and asynchronous capabilities for joining with the operation, retrieving its results, and so on, you can build useful libraries of combinators that compose tasks to build larger patterns. As discussed in the previous section, the .NET Framework includes several built-in combinators, but you can also build your own. The following sections provide several examples of potential combinator methods and types.

RetryOnFault

In many situations, you may want to retry an operation if a previous attempt fails. For synchronous code, you might build a helper method such as `RetryOnFault` in the following example to accomplish this:

```
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);
}
```

You can build an almost identical helper method for asynchronous operations that are implemented with TAP and thus return 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);
}
```

You can then use this combinator to encode retries into the application's logic; for example:

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

You could extend the `RetryOnFault` function further. For example, the function could accept another `Func<Task>` that will be invoked between retries to determine when to try the operation again; for example:

```
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().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
        await retryWhen().ConfigureAwait(false);
    }
    return default(T);
}
```

You could then use the function as follows to wait for a second before retrying the operation:

```
// 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, you can take advantage of redundancy to improve an operation's latency and chances for success. Consider multiple web services that provide stock quotes, but at various times of the day, each service may provide different levels of quality and response times. To deal with these fluctuations, you may issue requests to all the web services, and as soon as you get a response from one, cancel the remaining requests. You can implement a helper function to make it easier to implement this common pattern of launching multiple operations, waiting for any, and then canceling the rest. The `NeedOnlyOne` function in the following example illustrates this scenario:

```
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;
}
```

You can then use this function as follows:

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

Interleaved Operations

There is a potential performance problem with using the WhenAny method to support an interleaving scenario when you're working with very large sets of tasks. Every call to WhenAny results in a continuation being registered with each task. For N number of tasks, this results in O(N2) continuations created over the lifetime of the interleaving operation. If you're working with a large set of tasks, you can use a combinator (`Interleaved` in the following example) to address the performance issue:

```
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;
}
```

You can then use the combinator to process the results of tasks as they complete; for example:

```
IEnumerable<Task<int>> tasks = ...;
foreach(var task in Interleaved(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. You can accomplish that with a combinator method such as `WhenAllOrFirstException` in the following 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 and 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 a task is that it may be handed out to multiple consumers, all of whom may await it, register continuations with it, get its result or exceptions (in the case of Task<TResult>), and so on. This makes Task and Task<TResult> perfectly suited to be used in an asynchronous caching infrastructure. Here's an example of 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<TResult>. Any previously accessed values from the cache are stored in the internal dictionary, and the `AsyncCache` ensures that only one task is generated per key, even if the cache is accessed concurrently.

For example, you can build a cache for downloaded web pages:

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

You can then use this cache in asynchronous methods whenever you need the contents of a web page. The `AsyncCache` class ensures that you're downloading as few pages as possible, and caches 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

You can also use tasks to build data structures for coordinating asynchronous activities. Consider one of the classic parallel design patterns: producer/consumer. In this pattern, producers generate data that is consumed by

consumers, and the producers and consumers may run in parallel. For example, the consumer processes item 1, which was previously generated by a producer who is now producing item 2. For the producer/consumer pattern, you 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 data structure in place, you can write code such as 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);
}
```

The System.Threading.Tasks.Dataflow namespace includes the BufferBlock<T> type, which you can use in a similar manner, but 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);
}
```

NOTE

The System.Threading.Tasks.Dataflow namespace is available in the .NET Framework 4.5 through **NuGet**. To install the assembly that contains the System.Threading.Tasks.Dataflow namespace, open your project in Visual Studio 2012, choose **Manage NuGet Packages** from the Project menu, and search online for the Microsoft.Tpl.Dataflow package.

See Also

Task-based Asynchronous Pattern (TAP)
Implementing the Task-based Asynchronous Pattern
Interop with Other Asynchronous Patterns and Types

Interop with Other Asynchronous Patterns and Types

11/21/2017 • 6 min to read • Edit Online

The .NET Framework 1.0 introduced the IAsyncResult pattern, otherwise known as the Asynchronous Programming Model (APM), or the `Begin/End` pattern. The .NET Framework 2.0 added the Event-based Asynchronous Pattern (EAP). Starting with the .NET Framework 4, the Task-based Asynchronous Pattern (TAP) supersedes both APM and EAP, but provides the ability to easily build migration routines from the earlier patterns.

In this topic:

Tasks and the Asynchronous Programming Model (APM)

From APM to TAP

Because the Asynchronous Programming Model (APM) pattern is very structured, it is quite easy to build a wrapper to expose an APM implementation as a TAP implementation. In fact, the .NET Framework, starting with .NET Framework 4, includes helper routines in the form of FromAsync method overloads to provide this translation.

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

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

```
Public Function Read(buffer As Byte(), offset As Integer,
                     count As Integer) As Integer
```

```
public IAsyncResult BeginRead(byte[] buffer, int offset,
                              int count, AsyncCallback callback,
                              object state)
```

```
Public Function BeginRead(buffer As Byte, offset As Integer,
                          count As Integer, callback As AsyncCallback,
                          state As Object) As IAsyncResult
```

```
public int EndRead(IAsyncResult asyncResult)
```

```
Public Function EndRead(asyncResult As IAsyncResult) As Integer
```

You can use the TaskFactory<TResult>.FromAsync method to implement a TAP wrapper for this operation 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);
}
```

```
<Extension()>
Public Function ReadAsync(strm As Stream,
                          buffer As Byte(), offset As Integer,
                          count As Integer) As Task(Of Integer)
    If strm Is Nothing Then
       Throw New ArgumentNullException("stream")
    End If

    Return Task(Of Integer).Factory.FromAsync(AddressOf strm.BeginRead,
                                              AddressOf strm.EndRead, buffer,
                                              offset, count, Nothing)
End Function
```

This implementation is similar 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;
}
```

```
<Extension()>
Public Function ReadAsync(stream As Stream, buffer As Byte(), _
                          offset As Integer, count As Integer) _
                          As Task(Of Integer)
   If stream Is Nothing Then
       Throw New ArgumentNullException("stream")
   End If

   Dim tcs As New TaskCompletionSource(Of Integer)()
   stream.BeginRead(buffer, offset, count,
                    Sub(iar)
                       Try
                          tcs.TrySetResult(stream.EndRead(iar))
                       Catch e As OperationCanceledException
                          tcs.TrySetCanceled()
                       Catch e As Exception
                          tcs.TrySetException(e)
                       End Try
                    End Sub, Nothing)
   Return tcs.Task
End Function
```

From TAP to APM

If your existing infrastructure expects the APM pattern, you'll also want to take a TAP implementation and use it where an APM implementation is expected. Because tasks can be composed and the Task class implements IAsyncResult, you can use a straightforward helper function to do this. The following code uses an extension of the Task<TResult> class, but you can use an almost identical function for non-generic tasks.

```
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;
}
```

```
<Extension()>
Public Function AsApm(Of T)(task As Task(Of T),
                            callback As AsyncCallback,
                            state As Object) As IAsyncResult
    If task Is Nothing Then
        Throw New ArgumentNullException("task")
    End If

    Dim tcs As New TaskCompletionSource(Of T)(state)
    task.ContinueWith(Sub(antecedent)
                        If antecedent.IsFaulted Then
                           tcs.TrySetException(antecedent.Exception.InnerExceptions)
                        ElseIf antecedent.IsCanceled Then
                           tcs.TrySetCanceled()
                        Else
                           tcs.TrySetResult(antecedent.Result)
                        End If

                        If callback IsNot Nothing Then
                           callback(tcs.Task)
                        End If
                      End Sub, TaskScheduler.Default)
    Return tcs.Task
End Function
```

Now, consider a case where you have the following TAP implementation:

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

```
Public Shared Function DownloadStringAsync(url As Uri) As Task(Of String)
```

and you want to provide this APM implementation:

```
public IAsyncResult BeginDownloadString(Uri url,
                                        AsyncCallback callback,
                                        object state)
```

```
Public Function BeginDownloadString(url As Uri,
                                    callback As AsyncCallback,
                                    state As Object) As IAsyncResult
```

```
public string EndDownloadString(IAsyncResult asyncResult)
```

```
Public Function EndDownloadString(asyncResult As IAsyncResult) As String
```

The following example demonstrates one migration to APM:

```
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;
}
```

```
Public Function BeginDownloadString(url As Uri,
                                    callback As AsyncCallback,
                                    state As Object) As IAsyncResult
   Return DownloadStringAsync(url).AsApm(callback, state)
End Function

Public Function EndDownloadString(asyncResult As IAsyncResult) As String
   Return CType(asyncResult, Task(Of String)).Result
End Function
```

Tasks and the Event-based Asynchronous Pattern (EAP)

Wrapping an Event-based Asynchronous Pattern (EAP) implementation is more involved than wrapping an APM pattern, because the EAP pattern has more variation and less structure than the APM pattern. To demonstrate, the following code wraps 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 it's done. The final result is a string that contains 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;
 }
```

```
 Public Shared Function DownloadStringAsync(url As Uri) As Task(Of String)
     Dim tcs As New TaskCompletionSource(Of String)()
     Dim wc As New WebClient()
     AddHandler wc.DownloadStringCompleted, Sub(s,e)
             If e.Error IsNot Nothing Then
                tcs.TrySetException(e.Error)
             ElseIf e.Cancelled Then
                tcs.TrySetCanceled()
             Else
                tcs.TrySetResult(e.Result)
             End If
         End Sub
     wc.DownloadStringAsync(url)
     Return tcs.Task
End Function
```

Tasks and Wait Handles

From Wait Handles to TAP

Although wait handles don't implement an asynchronous pattern, advanced developers may use the WaitHandle class and the ThreadPool.RegisterWaitForSingleObject method for asynchronous notifications when a wait handle is set. You can wrap the RegisterWaitForSingleObject method to enable a task-based alternative to any synchronous wait on a wait handle:

```
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( (antecedent) => rwh.Unregister(null));
    return t;
}
```

```
<Extension()>
Public Function WaitOneAsync(waitHandle As WaitHandle) As Task
    If waitHandle Is Nothing Then
        Throw New ArgumentNullException("waitHandle")
    End If

    Dim tcs As New TaskCompletionSource(Of Boolean)()
    Dim rwh As RegisteredWaitHandle = ThreadPool.RegisterWaitForSingleObject(waitHandle,
        Sub(state, timedOut)
           tcs.TrySetResult(True)
        End Sub, Nothing, -1, True)
    Dim t = tcs.Task
    t.ContinueWith( Sub(antecedent)
                       rwh.Unregister(Nothing)
                   End Sub)
    Return t
End Function
```

With this method, you can use existing WaitHandle implementations in asynchronous methods. For example, if you want to throttle the number of asynchronous operations that are executing at any particular time, you can utilize a semaphore (a System.Threading.SemaphoreSlim object). You can throttle to *N* the number of operations that run concurrently by initializing the semaphore's count to *N*, waiting on the semaphore any time you want to

perform an operation, and releasing the semaphore when you're done with an operation:

```
static int N = 3;

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

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

```
Shared N As Integer = 3

Shared m_throttle As New SemaphoreSlim(N, N)

Shared Async Function DoOperation() As Task
    Await m_throttle.WaitAsync()
    ' Do work.
    m_throttle.Release()
End Function
```

You can also build an asynchronous semaphore that does not rely on wait handles and instead works completely with tasks. To do this, you can use techniques such as those discussed in Consuming the Task-based Asynchronous Pattern for building data structures on top of Task.

From TAP to Wait Handles

As previously mentioned, the Task class implements IAsyncResult, and that implementation exposes an IAsyncResult.AsyncWaitHandle property that returns a wait handle that will be set when the Task completes. You can get a WaitHandle for a Task as follows:

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

```
Dim wh As WaitHandle = CType(task, IAsyncResult).AsyncWaitHandle
```

See Also

Task-based Asynchronous Pattern (TAP)
Implementing the Task-based Asynchronous Pattern
Consuming the Task-based Asynchronous Pattern

Event-based Asynchronous Pattern (EAP)

11/21/2017 • 1 min to read • Edit Online

There are a number of ways to expose asynchronous features to client code. The Event-based Asynchronous Pattern prescribes one way for classes to present asynchronous behavior.

> **NOTE**
> Starting with the .NET Framework 4, the Task Parallel Library provides a new model for asynchronous and parallel programming. For more information, see Parallel Programming.

In This Section

Event-based Asynchronous Pattern Overview
Describes how the Event-based Asynchronous Pattern makes available the advantages of multithreaded applications while hiding many of the complex issues inherent in multithreaded design.

Implementing the Event-based Asynchronous Pattern
Describes the standardized way to package a class that has asynchronous features.

Best Practices for Implementing the Event-based Asynchronous Pattern
Describes the requirements for exposing asynchronous features according to the Event-based Asynchronous Pattern.

Deciding When to Implement the Event-based Asynchronous Pattern
Describes how to determine when you should choose to implement the Event-based Asynchronous Pattern instead of the IAsyncResult pattern.

Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern
Illustrates how to create a component that implements the Event-based Asynchronous Pattern. It is implemented using helper classes from the System.ComponentModel namespace, which ensures that the component works correctly under any application model.

How to: Use Components That Support the Event-based Asynchronous Pattern
Describes how to use a component that supports the Event-based Asynchronous Pattern.

Reference

AsyncOperation
Describes the AsyncOperation class and has links to all its members.

AsyncOperationManager
Describes the AsyncOperationManager class and has links to all its members.

BackgroundWorker
Describes the BackgroundWorker component and has links to all its members.

Related Sections

Task Parallel Library (TPL)
Describes a programming model for asynchronous and parallel operations.

Threading
Describes multithreading features in the .NET Framework.

Threading
Describes multithreading features in the C# and Visual Basic languages.

See Also

Managed Threading Best Practices
Events
Multithreading in Components
Asynchronous Programming Design Patterns

Multithreaded Programming with the Event-based Asynchronous Pattern

11/21/2017 • 1 min to read • Edit Online

There are a number of ways to expose asynchronous features to client code. The Event-based Asynchronous Pattern prescribes the recommended way for classes to present asynchronous behavior.

In This Section

Event-based Asynchronous Pattern Overview
Describes how the Event-based Asynchronous Pattern makes available the advantages of multithreaded applications while hiding many of the complex issues inherent in multithreaded design.

Implementing the Event-based Asynchronous Pattern
Describes the standardized way to package a class that has asynchronous features.

Best Practices for Implementing the Event-based Asynchronous Pattern
Describes the requirements for exposing asynchronous features according to the Event-based Asynchronous Pattern.

Deciding When to Implement the Event-based Asynchronous Pattern
Describes how to determine when you should choose to implement the Event-based Asynchronous Pattern instead of the IAsyncResult pattern.

Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern
Illustrates how to create a component that implements the Event-based Asynchronous Pattern. It is implemented using helper classes from the System.ComponentModel namespace, which ensures that the component works correctly under any application model.

How to: Use Components That Support the Event-based Asynchronous Pattern
Describes how to use a component that supports the Event-based Asynchronous Pattern.

Reference

AsyncOperation
Describes the AsyncOperation class and has links to all its members.

AsyncOperationManager
Describes the AsyncOperationManager class and has links to all its members.

BackgroundWorker
Describes the BackgroundWorker component and has links to all its members.

See Also

Managed Threading Best Practices
Events
Multithreading in Components
Event-based Asynchronous Pattern (EAP)

Event-based Asynchronous Pattern Overview

11/21/2017 • 6 min to read • Edit Online

Applications that perform many tasks simultaneously, yet remain responsive to user interaction, often require a design that uses multiple threads. The System.Threading namespace provides all the tools necessary to create high-performance multithreaded applications, but using these tools effectively requires significant experience with multithreaded software engineering. For relatively simple multithreaded applications, the BackgroundWorker component provides a straightforward solution. For more sophisticated asynchronous applications, consider implementing a class that adheres to the Event-based Asynchronous Pattern.

The Event-based Asynchronous Pattern makes available the advantages of multithreaded applications while hiding many of the complex issues inherent in multithreaded design. Using a class that supports this pattern can allow you to:

- Perform time-consuming tasks, such as downloads and database operations, "in the background," without interrupting your application.
- Execute multiple operations simultaneously, receiving notifications when each completes.
- Wait for resources to become available without stopping ("hanging") your application.
- Communicate with pending asynchronous operations using the familiar events-and-delegates model. For more information on using event handlers and delegates, see Events.

A class that supports the Event-based Asynchronous Pattern will have one or more methods named *MethodName* `Async`. These methods may mirror synchronous versions, which perform the same operation on the current thread. The class may also have a *MethodName* `Completed` event and it may have a *MethodName* `AsyncCancel` (or simply `CancelAsync`) method.

PictureBox is a typical component that supports the Event-based Asynchronous Pattern. You can download an image synchronously by calling its Load method, but if the image is large, or if the network connection is slow, your application will stop ("hang") until the download operation is completed and the call to Load returns.

If you want your application to keep running while the image is loading, you can call the LoadAsync method and handle the LoadCompleted event, just as you would handle any other event. When you call the LoadAsync method, your application will continue to run while the download proceeds on a separate thread ("in the background"). Your event handler will be called when the image-loading operation is complete, and your event handler can examine the AsyncCompletedEventArgs parameter to determine if the download completed successfully.

The Event-based Asynchronous Pattern requires that an asynchronous operation can be canceled, and the PictureBox control supports this requirement with its CancelAsync method. Calling CancelAsync submits a request to stop the pending download, and when the task is canceled, the LoadCompleted event is raised.

Caution

It is possible that the download will finish just as the CancelAsync request is made, so Cancelled may not reflect the request to cancel. This is called a *race condition* and is a common issue in multithreaded programming. For more information on issues in multithreaded programming, see Managed Threading Best Practices.

Characteristics of the Event-based Asynchronous Pattern

The Event-based Asynchronous Pattern may take several forms, depending on the complexity of the operations supported by a particular class. The simplest classes may have a single *MethodName* `Async` method and a

corresponding *MethodName* `Completed` event. More complex classes may have several *MethodName* `Async` methods, each with a corresponding *MethodName* `Completed` event, as well as synchronous versions of these methods. Classes can optionally support cancellation, progress reporting, and incremental results for each asynchronous method.

An asynchronous method may also support multiple pending calls (multiple concurrent invocations), allowing your code to call it any number of times before it completes other pending operations. Correctly handling this situation may require your application to track the completion of each operation.

Examples of the Event-based Asynchronous Pattern

The SoundPlayer and PictureBox components represent simple implementations of the Event-based Asynchronous Pattern. The WebClient and BackgroundWorker components represent more complex implementations of the Event-based Asynchronous Pattern.

Below is an example class declaration that conforms to the pattern:

```
Public Class AsyncExample
    ' Synchronous methods.
    Public Function Method1(ByVal param As String) As Integer
    Public Sub Method2(ByVal param As Double)

    ' Asynchronous methods.
    Overloads Public Sub Method1Async(ByVal param As String)
    Overloads Public Sub Method1Async(ByVal param As String, ByVal userState As Object)
    Public Event Method1Completed As Method1CompletedEventHandler

    Overloads Public Sub Method2Async(ByVal param As Double)
    Overloads Public Sub Method2Async(ByVal param As Double, ByVal userState As Object)
    Public Event Method2Completed As Method2CompletedEventHandler

    Public Sub CancelAsync(ByVal userState As Object)

    Public ReadOnly Property IsBusy () As Boolean

    ' Class implementation not shown.
End Class
```

```
public class AsyncExample
{
    // Synchronous methods.
    public int Method1(string param);
    public void Method2(double param);

    // Asynchronous methods.
    public void Method1Async(string param);
    public void Method1Async(string param, object userState);
    public event Method1CompletedEventHandler Method1Completed;

    public void Method2Async(double param);
    public void Method2Async(double param, object userState);
    public event Method2CompletedEventHandler Method2Completed;

    public void CancelAsync(object userState);

    public bool IsBusy { get; }

    // Class implementation not shown.
}
```

The fictitious `AsyncExample` class has two methods, both of which support synchronous and asynchronous invocations. The synchronous overloads behave like any method call and execute the operation on the calling

thread; if the operation is time-consuming, there may be a noticeable delay before the call returns. The asynchronous overloads will start the operation on another thread and then return immediately, allowing the calling thread to continue while the operation executes "in the background."

Asynchronous Method Overloads

There are potentially two overloads for the asynchronous operations: single-invocation and multiple-invocation. You can distinguish these two forms by their method signatures: the multiple-invocation form has an extra parameter called `userState`. This form makes it possible for your code to call `Method1Async(string param, object userState)` multiple times without waiting for any pending asynchronous operations to finish. If, on the other hand, you try to call `Method1Async(string param)` before a previous invocation has completed, the method raises an InvalidOperationException.

The `userState` parameter for the multiple-invocation overloads allows you to distinguish among asynchronous operations. You provide a unique value (for example, a GUID or hash code) for each call to `Method1Async(string param, object userState)`, and when each operation is completed, your event handler can determine which instance of the operation raised the completion event.

Tracking Pending Operations

If you use the multiple-invocation overloads, your code will need to keep track of the `userState` objects (task IDs) for pending tasks. For each call to `Method1Async(string param, object userState)`, you will typically generate a new, unique `userState` object and add it to a collection. When the task corresponding to this `userState` object raises the completion event, your completion method implementation will examine AsyncCompletedEventArgs.UserState and remove it from your collection. Used this way, the `userState` parameter takes the role of a task ID.

> **NOTE**
> You must be careful to provide a unique value for `userState` in your calls to multiple-invocation overloads. Non-unique task IDs will cause the asynchronous class throw an ArgumentException.

Canceling Pending Operations

It is important to be able to cancel asynchronous operations at any time before their completion. Classes that implement the Event-based Asynchronous Pattern will have a `CancelAsync` method (if there is only one asynchronous method) or a *MethodName* `AsyncCancel` method (if there are multiple asynchronous methods).

Methods that allow multiple invocations take a `userState` parameter, which can be used to track the lifetime of each task. `CancelAsync` takes a `userState` parameter, which allows you to cancel particular pending tasks.

Methods that support only a single pending operation at a time, like `Method1Async(string param)`, are not cancelable.

Receiving Progress Updates and Incremental Results

A class that adheres to the Event-based Asynchronous Pattern may optionally provide an event for tracking progress and incremental results. This will typically be named `ProgressChanged` or *MethodName* `ProgressChanged`, and its corresponding event handler will take a ProgressChangedEventArgs parameter.

The event handler for the `ProgressChanged` event can examine the ProgressChangedEventArgs.ProgressPercentage property to determine what percentage of an asynchronous task has been completed. This property will range from 0 to 100, and it can be used to update the Value property of a ProgressBar. If multiple asynchronous operations are pending, you can use the ProgressChangedEventArgs.UserState property to distinguish which operation is reporting progress.

Some classes may report incremental results as asynchronous operations proceed. These results will be stored in a class that derives from ProgressChangedEventArgs and they will appear as properties in the derived class. You

can access these results in the event handler for the `ProgressChanged` event, just as you would access the ProgressPercentage property. If multiple asynchronous operations are pending, you can use the UserState property to distinguish which operation is reporting incremental results.

See Also

ProgressChangedEventArgs
BackgroundWorker
AsyncCompletedEventArgs
How to: Use Components That Support the Event-based Asynchronous Pattern
How to: Run an Operation in the Background
How to: Implement a Form That Uses a Background Operation
Multithreaded Programming with the Event-based Asynchronous Pattern
Best Practices for Implementing the Event-based Asynchronous Pattern
Deciding When to Implement the Event-based Asynchronous Pattern

Implementing the Event-based Asynchronous Pattern

11/21/2017 • 8 min to read • Edit Online

If you are writing a class with some operations that may incur noticeable delays, consider giving it asynchronous functionality by implementing Event-based Asynchronous Pattern Overview.

The Event-based Asynchronous Pattern provides a standardized way to package a class that has asynchronous features. If implemented with helper classes like AsyncOperationManager, your class will work correctly under any application model, including ASP.NET, Console applications, and Windows Forms applications.

For an example that implements the Event-based Asynchronous Pattern, see How to: Implement a Component That Supports the Event-based Asynchronous Pattern.

For simple asynchronous operations, you may find the BackgroundWorker component suitable. For more information about BackgroundWorker, see How to: Run an Operation in the Background.

The following list describes the features of the Event-based Asynchronous Pattern discussed in this topic.

- Opportunities for Implementing the Event-based Asynchronous Pattern
- Naming Asynchronous Methods
- Optionally Support Cancellation
- Optionally Support the IsBusy Property
- Optionally Provide Support for Progress Reporting
- Optionally Provide Support for Returning Incremental Results
- Handling Out and Ref Parameters in Methods

Opportunities for Implementing the Event-based Asynchronous Pattern

Consider implementing the Event-based Asynchronous Pattern when:

- Clients of your class do not need WaitHandle and IAsyncResult objects available for asynchronous operations, meaning that polling and WaitAll or WaitAny will need to be built up by the client.
- You want asynchronous operations to be managed by the client with the familiar event/delegate model.

Any operation is a candidate for an asynchronous implementation, but those you expect to incur long latencies should be considered. Especially appropriate are operations in which clients call a method and are notified on completion, with no further intervention required. Also appropriate are operations which run continuously, periodically notifying clients of progress, incremental results, or state changes.

For more information on deciding when to support the Event-based Asynchronous Pattern, see Deciding When to Implement the Event-based Asynchronous Pattern.

Naming Asynchronous Methods

For each synchronous method *MethodName* for which you want to provide an asynchronous counterpart:

Define a *MethodName* `Async` method that:

- Returns `void`.
- Takes the same parameters as the *MethodName* method.
- Accepts multiple invocations.

Optionally define a *MethodName* `Async` overload, identical to *MethodName* `Async`, but with an additional object-valued parameter called `userState`. Do this if you're prepared to manage multiple concurrent invocations of your method, in which case the `userState` value will be delivered back to all event handlers to distinguish invocations of the method. You may also choose to do this simply as a place to store user state for later retrieval.

For each separate *MethodName* `Async` method signature:

1. Define the following event in the same class as the method:

```
Public Event MethodNameCompleted As MethodNameCompletedEventHandler
```

```
public event MethodNameCompletedEventHandler MethodNameCompleted;
```

2. Define the following delegate and AsyncCompletedEventArgs. These will likely be defined outside of the class itself, but in the same namespace.

```
Public Delegate Sub MethodNameCompletedEventHandler( _
    ByVal sender As Object, _
    ByVal e As MethodNameCompletedEventArgs)

Public Class MethodNameCompletedEventArgs
    Inherits System.ComponentModel.AsyncCompletedEventArgs
Public ReadOnly Property Result() As MyReturnType
End Property
```

```
public delegate void MethodNameCompletedEventHandler(object sender,
    MethodNameCompletedEventArgs e);

public class MethodNameCompletedEventArgs : System.ComponentModel.AsyncCompletedEventArgs
{
    public MyReturnType Result { get; }
}
```

   - Ensure that the *MethodName* `CompletedEventArgs` class exposes its members as read-only properties, and not fields, as fields prevent data binding.
   - Do not define any AsyncCompletedEventArgs-derived classes for methods that do not produce results. Simply use an instance of AsyncCompletedEventArgs itself.

     NOTE

     It is perfectly acceptable, when feasible and appropriate, to reuse delegate and AsyncCompletedEventArgs types. In this case, the naming will not be as consistent with the method name, since a given delegate and AsyncCompletedEventArgs won't be tied to a single method.

Optionally Support Cancellation

If your class will support canceling asynchronous operations, cancellation should be exposed to the client as described below. Note that there are two decision points that need to be reached before defining your cancellation

support:

- Does your class, including future anticipated additions to it, have only one asynchronous operation that supports cancellation?
- Can the asynchronous operations that support cancellation support multiple pending operations? That is, does the *MethodName* `Async` method take a `userState` parameter, and does it allow multiple invocations before waiting for any to finish?

Use the answers to these two questions in the table below to determine what the signature for your cancellation method should be.

Visual Basic

	MULTIPLE SIMULTANEOUS OPERATIONS SUPPORTED	ONLY ONE OPERATION AT A TIME
One Async Operation in entire class	`Sub MethodNameAsyncCancel(ByVal userState As Object)`	`Sub MethodNameAsyncCancel()`
Multiple Async Operations in class	`Sub CancelAsync(ByVal userState As Object)`	`Sub CancelAsync()`

C#

	MULTIPLE SIMULTANEOUS OPERATIONS SUPPORTED	ONLY ONE OPERATION AT A TIME
One Async Operation in entire class	`void MethodNameAsyncCancel(object userState);`	`void MethodNameAsyncCancel();`
Multiple Async Operations in class	`void CancelAsync(object userState);`	`void CancelAsync();`

If you define the `CancelAsync(object userState)` method, clients must be careful when choosing their state values to make them capable of distinguishing among all asynchronous methods invoked on the object, and not just between all invocations of a single asynchronous method.

The decision to name the single-async-operation version *MethodName* `AsyncCancel` is based on being able to more easily discover the method in a design environment like Visual Studio's IntelliSense. This groups the related members and distinguishes them from other members that have nothing to do with asynchronous functionality. If you expect that there may be additional asynchronous operations added in subsequent versions, it is better to define `CancelAsync`.

Do not define multiple methods from the table above in the same class. That will not make sense, or it will clutter the class interface with a proliferation of methods.

These methods typically will return immediately, and the operation may or may not actually cancel. In the event handler for the *MethodName* `Completed` event, the *MethodName* `CompletedEventArgs` object contains a `Cancelled` field, which clients can use to determine whether the cancellation occurred.

Abide by the cancellation semantics described in Best Practices for Implementing the Event-based Asynchronous Pattern.

Optionally Support the IsBusy Property

If your class does not support multiple concurrent invocations, consider exposing an `IsBusy` property. This allows developers to determine whether a *MethodName* `Async` method is running without catching an exception from

the *MethodName* `Async` method.

Abide by the `IsBusy` semantics described in Best Practices for Implementing the Event-based Asynchronous Pattern.

Optionally Provide Support for Progress Reporting

It is frequently desirable for an asynchronous operation to report progress during its operation. The Event-based Asynchronous Pattern provides a guideline for doing so.

- Optionally define an event to be raised by the asynchronous operation and invoked on the appropriate thread. The ProgressChangedEventArgs object carries an integer-valued progress indicator that is expected to be between 0 and 100.
- Name this event as follows:
  - `ProgressChanged` if the class has multiple asynchronous operations (or is expected to grow to include multiple asynchronous operations in future versions);
  - *MethodName* `ProgressChanged` if the class has a single asynchronous operation.

  This naming choice parallels that made for the cancellation method, as described in the Optionally Support Cancellation section.

This event should use the ProgressChangedEventHandler delegate signature and the ProgressChangedEventArgs class. Alternatively, if a more domain-specific progress indicator can be provided (for instance, bytes read and total bytes for a download operation), then you should define a derived class of ProgressChangedEventArgs.

Note that there is only one `ProgressChanged` or *MethodName* `ProgressChanged` event for the class, regardless of the number of asynchronous methods it supports. Clients are expected to use the `userState` object that is passed to the *MethodName* `Async` methods to distinguish among progress updates on multiple concurrent operations.

There may be situations in which multiple operations support progress and each returns a different indicator for progress. In this case, a single `ProgressChanged` event is not appropriate, and you may consider supporting multiple `ProgressChanged` events. In this case use a naming pattern of *MethodName* `ProgressChanged` for each *MethodName* `Async` method.

Abide by the progress-reporting semantics described Best Practices for Implementing the Event-based Asynchronous Pattern.

Optionally Provide Support for Returning Incremental Results

Sometimes an asynchronous operation can return incremental results prior to completion. There are a number of options that can be used to support this scenario. Some examples follow.

Single-operation Class

If your class only supports a single asynchronous operation, and that operation is able to return incremental results, then:

- Extend the ProgressChangedEventArgs type to carry the incremental result data, and define a *MethodName* `ProgressChanged` event with this extended data.
- Raise this *MethodName* `ProgressChanged` event when there is an incremental result to report.

This solution applies specifically to a single-async-operation class because there is no problem with the same event occurring to return incremental results on "all operations", as the *MethodName* `ProgressChanged` event does.

Multiple-operation Class with Homogeneous Incremental Results

In this case, your class supports multiple asynchronous methods, each capable of returning incremental results, and these incremental results all have the same type of data.

Follow the model described above for single-operation classes, as the same EventArgs structure will work for all incremental results. Define a `ProgressChanged` event instead of a *MethodName* `ProgressChanged` event, since it applies to multiple asynchronous methods.

Multiple-operation Class with Heterogeneous Incremental Results

If your class supports multiple asynchronous methods, each returning a different type of data, you should:

- Separate your incremental result reporting from your progress reporting.
- Define a separate *MethodName* `ProgressChanged` event with appropriate EventArgs for each asynchronous method to handle that method's incremental result data.

Invoke that event handler on the appropriate thread as described in Best Practices for Implementing the Event-based Asynchronous Pattern.

Handling Out and Ref Parameters in Methods

Although the use of `out` and `ref` is, in general, discouraged in the .NET Framework, here are the rules to follow when they are present:

Given a synchronous method *MethodName*:

- `out` parameters to *MethodName* should not be part of *MethodName* `Async`. Instead, they should be part of *MethodName* `CompletedEventArgs` with the same name as its parameter equivalent in *MethodName* (unless there is a more appropriate name).
- `ref` parameters to *MethodName* should appear as part of *MethodName* `Async`, and as part of *MethodName* `CompletedEventArgs` with the same name as its parameter equivalent in *MethodName* (unless there is a more appropriate name).

For example, given:

```
Public Function MethodName(ByVal arg1 As String, ByRef arg2 As String, ByRef arg3 As String) As Integer
```

```
public int MethodName(string arg1, ref string arg2, out string arg3);
```

Your asynchronous method and its AsyncCompletedEventArgs class would look like this:

```
Public Sub MethodNameAsync(ByVal arg1 As String, ByVal arg2 As String)

Public Class MethodNameCompletedEventArgs
    Inherits System.ComponentModel.AsyncCompletedEventArgs
    Public ReadOnly Property Result() As Integer
    End Property
    Public ReadOnly Property Arg2() As String
    End Property
    Public ReadOnly Property Arg3() As String
    End Property
End Class
```

```
public void MethodNameAsync(string arg1, string arg2);

public class MethodNameCompletedEventArgs : System.ComponentModel.AsyncCompletedEventArgs
{
    public int Result { get; };
    public string Arg2 { get; };
    public string Arg3 { get; };
}
```

See Also

ProgressChangedEventArgs
AsyncCompletedEventArgs
How to: Implement a Component That Supports the Event-based Asynchronous Pattern
How to: Run an Operation in the Background
How to: Implement a Form That Uses a Background Operation
Deciding When to Implement the Event-based Asynchronous Pattern
Multithreaded Programming with the Event-based Asynchronous Pattern
Best Practices for Implementing the Event-based Asynchronous Pattern

Best Practices for Implementing the Event-based Asynchronous Pattern

11/21/2017 • 8 min to read • Edit Online

The Event-based Asynchronous Pattern provides you with an effective way to expose asynchronous behavior in classes, with familiar event and delegate semantics. To implement Event-based Asynchronous Pattern, you need to follow some specific behavioral requirements. The following sections describe requirements and guidelines you should consider when you implement a class that follows the Event-based Asynchronous Pattern.

For an overview, see Implementing the Event-based Asynchronous Pattern.

Required Behavioral Guarantees

If you implement the Event-based Asynchronous Pattern, you must provide a number of guarantees to ensure that your class will behave properly and clients of your class can rely on such behavior.

Completion

Always invoke the *MethodName* `Completed` event handler when you have successful completion, an error, or a cancellation. Applications should never encounter a situation where they remain idle and completion never occurs. One exception to this rule is if the asynchronous operation itself it designed so that it never completes.

Completed Event and EventArgs

For each separate *MethodName* `Async` method, apply the following design requirements:

- Define a *MethodName* `Completed` event on the same class as the method.
- Define an EventArgs class and accompanying delegate for the *MethodName* `Completed` event that derives from the AsyncCompletedEventArgs class. The default class name should be of the form *MethodName* `CompletedEventArgs`.
- Ensure that the EventArgs class is specific to the return values of the *MethodName* method. When you use the EventArgs class, you should never require developers to cast the result.

  The following code example shows good and bad implementation of this design requirement respectively.

```
// Good design
private void Form1_MethodNameCompleted(object sender, xxxCompletedEventArgs e)
{
    DemoType result = e.Result;
}

// Bad design
private void Form1_MethodNameCompleted(object sender, MethodNameCompletedEventArgs e)
{
    DemoType result = (DemoType)(e.Result);
}
```

- Do not define an EventArgs class for returning methods that return `void`. Instead, use an instance of the AsyncCompletedEventArgs class.
- Ensure that you always raise the *MethodName* `Completed` event. This event should be raised on successful completion, on an error, or on cancellation. Applications should never encounter a situation where they remain idle and completion never occurs.

- Ensure that you catch any exceptions that occur in the asynchronous operation and assign the caught exception to the Error property.
- If there was an error completing the task, the results should not be accessible. When the Error property is not `null`, ensure that accessing any property in the EventArgs structure raises an exception. Use the RaiseExceptionIfNecessary method to perform this verification.
- Model a time out as an error. When a time out occurs, raise the *MethodName* `Completed` event and assign a TimeoutException to the Error property.
- If your class supports multiple concurrent invocations, ensure that the *MethodName* `Completed` event contains the appropriate `userSuppliedState` object.
- Ensure that the *MethodName* `Completed` event is raised on the appropriate thread and at the appropriate time in the application lifecycle. For more information, see the Threading and Contexts section.

Simultaneously Executing Operations

- If your class supports multiple concurrent invocations, enable the developer to track each invocation separately by defining the *MethodName* `Async` overload that takes an object-valued state parameter, or task ID, called `userSuppliedState`. This parameter should always be the last parameter in the *MethodName* `Async` method's signature.
- If your class defines the *MethodName* `Async` overload that takes an object-valued state parameter, or task ID, be sure to track the lifetime of the operation with that task ID, and be sure to provide it back into the completion handler. There are helper classes available to assist. For more information on concurrency management, see Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern.
- If your class defines the *MethodName* `Async` method without the state parameter, and it does not support multiple concurrent invocations, ensure that any attempt to invoke *MethodName* `Async` before the prior *MethodName* `Async` invocation has completed raises an InvalidOperationException.
- In general, do not raise an exception if the *MethodName* `Async` method without the `userSuppliedState` parameter is invoked multiple times so that there are multiple outstanding operations. You can raise an exception when your class explicitly cannot handle that situation, but assume that developers can handle these multiple indistinguishable callbacks

Accessing Results

- If there was an error during execution of the asynchronous operation, the results should not be accessible. Ensure that accessing any property in the AsyncCompletedEventArgs when Error is not `null` raises the exception referenced by Error. The AsyncCompletedEventArgs class provides the RaiseExceptionIfNecessary method for this purpose.
- Ensure that any attempt to access the result raises an InvalidOperationException stating that the operation was canceled. Use the AsyncCompletedEventArgs.RaiseExceptionIfNecessary method to perform this verification.

Progress Reporting

- Support progress reporting, if possible. This enables developers to provide a better application user experience when they use your class.
- If you implement a `ProgressChanged` /*MethodName* `ProgressChanged` event, ensure that there are no such events raised for a particular asynchronous operation after that operation's *MethodName* `Completed` event has been raised.
- If the standard ProgressChangedEventArgs is being populated, ensure that the ProgressPercentage can

always be interpreted as a percentage. The percentage does not need to be accurate, but it should represent a percentage. If your progress reporting metric must be something other than a percentage, derive a class from the ProgressChangedEventArgs class and leave ProgressPercentage at 0. Avoid using a reporting metric other than a percentage.

- Ensure that the `ProgressChanged` event is raised on the appropriate thread and at the appropriate time in the application lifecycle. For more information, see the Threading and Contexts section.

IsBusy Implementation

- Do not expose an `IsBusy` property if your class supports multiple concurrent invocations. For example, XML Web service proxies do not expose an `IsBusy` property because they support multiple concurrent invocations of asynchronous methods.

- The `IsBusy` property should return `true` after the *MethodName* `Async` method has been called and before the *MethodName* `Completed` event has been raised. Otherwise it should return `false`. The BackgroundWorker and WebClient components are examples of classes that expose an `IsBusy` property.

Cancellation

- Support cancellation, if possible. This enables developers to provide a better application user experience when they use your class.

- In the case of cancellation, set the Cancelled flag in the AsyncCompletedEventArgs object.

- Ensure that any attempt to access the result raises an InvalidOperationException stating that the operation was canceled. Use the AsyncCompletedEventArgs.RaiseExceptionIfNecessary method to perform this verification.

- Ensure that calls to a cancellation method always return successfully, and never raise an exception. In general, a client is not notified as to whether an operation is truly cancelable at any given time, and is not notified as to whether a previously issued cancellation has succeeded. However, the application will always be given notification when a cancellation succeeded, because the application takes part in the completion status.

- Raise the *MethodName* `Completed` event when the operation is canceled.

Errors and Exceptions

- Catch any exceptions that occur in the asynchronous operation and set the value of the AsyncCompletedEventArgs.Error property to that exception.

Threading and Contexts

For correct operation of your class, it is critical that the client's event handlers are invoked on the proper thread or context for the given application model, including ASP.NET and Windows Forms applications. Two important helper classes are provided to ensure that your asynchronous class behaves correctly under any application model: AsyncOperation and AsyncOperationManager.

AsyncOperationManager provides one method, CreateOperation, which returns an AsyncOperation. Your *MethodName* `Async` method calls CreateOperation and your class uses the returned AsyncOperation to track the lifetime of the asynchronous task.

To report progress, incremental results, and completion to the client, call the Post and OperationCompleted methods on the AsyncOperation. AsyncOperation is responsible for marshaling calls to the client's event handlers to the proper thread or context.

NOTE

You can circumvent these rules if you explicitly want to go against the policy of the application model, but still benefit from the other advantages of using the Event-based Asynchronous Pattern. For example, you may want a class operating in Windows Forms to be free threaded. You can create a free threaded class, as long as developers understand the implied restrictions. Console applications do not synchronize the execution of Post calls. This can cause `ProgressChanged` events to be raised out of order. If you wish to have serialized execution of Post calls, implement and install a System.Threading.SynchronizationContext class.

For more information about using AsyncOperation and AsyncOperationManager to enable your asynchronous operations, see Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern.

Guidelines

- Ideally, each method invocation should be independent of others. You should avoid coupling invocations with shared resources. If resources are to be shared among invocations, you will need to provide a proper synchronization mechanism in your implementation.
- Designs that require the client to implement synchronization are discouraged. For example, you could have an asynchronous method that receives a global static object as a parameter; multiple concurrent invocations of such a method could result in data corruption or deadlocks.
- If you implement a method with the multiple-invocation overload (`userState` in the signature), your class will need to manage a collection of user states, or task IDs, and their corresponding pending operations. This collection should be protected with `lock` regions, because the various invocations add and remove `userState` objects in the collection.
- Consider reusing `CompletedEventArgs` classes where feasible and appropriate. In this case, the naming is not consistent with the method name, because a given delegate and EventArgs type are not tied to a single method. However, forcing developers to cast the value retrieved from a property on the EventArgs is never acceptable.
- If you are authoring a class that derives from Component, do not implement and install your own SynchronizationContext class. Application models, not components, control the SynchronizationContext that is used.
- When you use multithreading of any sort, you potentially expose yourself to very serious and complex bugs. Before implementing any solution that uses multithreading, see Managed Threading Best Practices.

See Also

AsyncOperation
AsyncOperationManager
AsyncCompletedEventArgs
ProgressChangedEventArgs
BackgroundWorker
Implementing the Event-based Asynchronous Pattern
Multithreaded Programming with the Event-based Asynchronous Pattern
Deciding When to Implement the Event-based Asynchronous Pattern
Best Practices for Implementing the Event-based Asynchronous Pattern
How to: Use Components That Support the Event-based Asynchronous Pattern
Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern

Deciding When to Implement the Event-based Asynchronous Pattern

11/21/2017 • 3 min to read • Edit Online

The Event-based Asynchronous Pattern provides a pattern for exposing the asynchronous behavior of a class. With the introduction of this pattern, the .NET Framework defines two patterns for exposing asynchronous behavior: the Asynchronous Pattern based on the System.IAsyncResult interface, and the event-based pattern. This topic describes when it is appropriate for you to implement both patterns.

For more information about asynchronous programming with the IAsyncResult interface, see Event-based Asynchronous Pattern (EAP).

General Principles

In general, you should expose asynchronous features using the Event-based Asynchronous Pattern whenever possible. However, there are some requirements that the event-based pattern cannot meet. In those cases, you may need to implement the IAsyncResult pattern in addition to the event-based pattern.

> **NOTE**
> It is rare for the IAsyncResult pattern to be implemented without the event-based pattern also being implemented.

Guidelines

The following list describes the guidelines for when you should implement Event-based Asynchronous Pattern:

- Use the event-based pattern as the default API to expose asynchronous behavior for your class.
- Do not expose the IAsyncResult pattern when your class is primarily used in a client application, for example Windows Forms.
- Only expose the IAsyncResult pattern when it is necessary for meeting your requirements. For example, compatibility with an existing API may require you to expose the IAsyncResult pattern.
- Do not expose the IAsyncResult pattern without also exposing the event-based pattern.
- If you must expose the IAsyncResult pattern, do so as an advanced option. For example, if you generate a proxy object, generate the event-based pattern by default, with an option to generate the IAsyncResult pattern.
- Build your event-based pattern implementation on your IAsyncResult pattern implementation.
- Avoid exposing both the event-based pattern and the IAsyncResult pattern on the same class. Expose the event-based pattern on "higher level" classes and the IAsyncResult pattern on "lower level" classes. For example, compare the event-based pattern on the WebClient component with the IAsyncResult pattern on the HttpRequest class.
    - Expose the event-based pattern and the IAsyncResult pattern on the same class when compatibility requires it. For example, if you have already released an API that uses the IAsyncResult pattern, you would need to retain the IAsyncResult pattern for backward compatibility.
    - Expose the event-based pattern and the IAsyncResult pattern on the same class if the resulting

object model complexity outweighs the benefit of separating the implementations. It is better to expose both patterns on a single class than to avoid exposing the event-based pattern.

- If you must expose both the event-based pattern and IAsyncResult pattern on a single class, use EditorBrowsableAttribute set to Advanced to mark the IAsyncResult pattern implementation as an advanced feature. This indicates to design environments, such as Visual Studio IntelliSense, not to display the IAsyncResult properties and methods. These properties and methods are still fully usable, but the developer working through IntelliSense has a clearer view of the API.

Criteria for Exposing the IAsyncResult Pattern in Addition to the Event-based Pattern

While the Event-based Asynchronous Pattern has many benefits under the previously mentioned scenarios, it does have some drawbacks, which you should be aware of if performance is your most important requirement.

There are three scenarios that the event-based pattern does not address as well as the IAsyncResult pattern:

- Blocking wait on one IAsyncResult
- Blocking wait on many IAsyncResult objects
- Polling for completion on the IAsyncResult

You can address these scenarios by using the event-based pattern, but doing so is more cumbersome than using the IAsyncResult pattern.

Developers often use the IAsyncResult pattern for services that typically have very high performance requirements. For example, the polling for completion scenario is a high-performance server technique.

Additionally, the event-based pattern is less efficient than the IAsyncResult pattern because it creates more objects, especially EventArgs, and because it synchronizes across threads.

The following list shows some recommendations to follow if you decide to use the IAsyncResult pattern:

- Only expose the IAsyncResult pattern when you specifically require support for WaitHandle or IAsyncResult objects.
- Only expose the IAsyncResult pattern when you have an existing API that uses the IAsyncResult pattern.
- If you have an existing API based on the IAsyncResult pattern, consider also exposing the event-based pattern in your next release.
- Only expose IAsyncResult pattern if you have high performance requirements which you have verified cannot be met by the event-based pattern but can be met by the IAsyncResult pattern.

See Also

Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern
Event-based Asynchronous Pattern (EAP)
Multithreaded Programming with the Event-based Asynchronous Pattern
Implementing the Event-based Asynchronous Pattern
Best Practices for Implementing the Event-based Asynchronous Pattern
Event-based Asynchronous Pattern Overview

Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern

11/21/2017 • 21 min to read • Edit Online

If you are writing a class with some operations that may incur noticeable delays, consider giving it asynchronous functionality by implementing the Event-based Asynchronous Pattern Overview.

This walkthrough illustrates how to create a component that implements the Event-based Asynchronous Pattern. It is implemented using helper classes from the System.ComponentModel namespace, which ensures that the component works correctly under any application model, including ASP.NET, Console applications and Windows Forms applications. This component is also designable with a PropertyGrid control and your own custom designers.

When you are through, you will have an application that computes prime numbers asynchronously. Your application will have a main user interface (UI) thread and a thread for each prime number calculation. Although testing whether a large number is prime can take a noticeable amount of time, the main UI thread will not be interrupted by this delay, and the form will be responsive during the calculations. You will be able to run as many calculations as you like concurrently and selectively cancel pending calculations.

Tasks illustrated in this walkthrough include:

- Creating the Component
- Defining Public Asynchronous Events and Delegates
- Defining Private Delegates
- Implementing Public Events
- Implementing the Completion Method
- Implementing the Worker Methods
- Implementing Start and Cancel Methods

To copy the code in this topic as a single listing, see How to: Implement a Component That Supports the Event-based Asynchronous Pattern.

Creating the Component

The first step is to create the component that will implement the Event-based Asynchronous Pattern.

To create the component

- Create a class called `PrimeNumberCalculator` that inherits from Component.

Defining Public Asynchronous Events and Delegates

Your component communicates to clients using events. The *MethodName* `Completed` event alerts clients to the completion of an asynchronous task, and the *MethodName* `ProgressChanged` event informs clients of the progress of an asynchronous task.

To define asynchronous events for clients of your component:

1. Import the System.Threading and System.Collections.Specialized namespaces at the top of your file.

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Globalization;
using System.Threading;
using System.Windows.Forms;
```

```
Imports System
Imports System.Collections
Imports System.Collections.Specialized
Imports System.ComponentModel
Imports System.Drawing
Imports System.Globalization
Imports System.Threading
Imports System.Windows.Forms
```

2. Before the `PrimeNumberCalculator` class definition, declare delegates for progress and completion events.

```
public delegate void ProgressChangedEventHandler(
    ProgressChangedEventArgs e);

public delegate void CalculatePrimeCompletedEventHandler(
    object sender,
    CalculatePrimeCompletedEventArgs e);
```

```
Public Delegate Sub ProgressChangedEventHandler( _
    ByVal e As ProgressChangedEventArgs)

Public Delegate Sub CalculatePrimeCompletedEventHandler( _
    ByVal sender As Object, _
    ByVal e As CalculatePrimeCompletedEventArgs)
```

3. In the `PrimeNumberCalculator` class definition, declare events for reporting progress and completion to clients.

```
public event ProgressChangedEventHandler ProgressChanged;
public event CalculatePrimeCompletedEventHandler CalculatePrimeCompleted;
```

```
Public Event ProgressChanged _
    As ProgressChangedEventHandler
Public Event CalculatePrimeCompleted _
    As CalculatePrimeCompletedEventHandler
```

4. After the `PrimeNumberCalculator` class definition, derive the `CalculatePrimeCompletedEventArgs` class for reporting the outcome of each calculation to the client's event handler for the `CalculatePrimeCompleted` .event. In addition to the `AsyncCompletedEventArgs` properties, this class enables the client to determine what number was tested, whether it is prime, and what the first divisor is if it is not prime.

```
public class CalculatePrimeCompletedEventArgs :
    AsyncCompletedEventArgs
{
    private int numberToTestValue = 0;
    private int firstDivisorValue = 1;
    private bool isPrimeValue;

    public CalculatePrimeCompletedEventArgs(
        int numberToTest,
        int firstDivisor,
        bool isPrime,
        Exception e,
        bool canceled,
        object state) : base(e, canceled, state)
    {
        this.numberToTestValue = numberToTest;
        this.firstDivisorValue = firstDivisor;
        this.isPrimeValue = isPrime;
    }

    public int NumberToTest
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return numberToTestValue;
        }
    }

    public int FirstDivisor
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return firstDivisorValue;
        }
    }

    public bool IsPrime
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return isPrimeValue;
        }
    }
}
```

```
Public Class CalculatePrimeCompletedEventArgs
    Inherits AsyncCompletedEventArgs
    Private numberToTestValue As Integer = 0
    Private firstDivisorValue As Integer = 1
    Private isPrimeValue As Boolean


    Public Sub New( _
    ByVal numberToTest As Integer, _
    ByVal firstDivisor As Integer, _
    ByVal isPrime As Boolean, _
    ByVal e As Exception, _
    ByVal canceled As Boolean, _
    ByVal state As Object)

        MyBase.New(e, canceled, state)
        Me.numberToTestValue = numberToTest
        Me.firstDivisorValue = firstDivisor
        Me.isPrimeValue = isPrime

    End Sub


    Public ReadOnly Property NumberToTest() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return numberToTestValue
        End Get
    End Property


    Public ReadOnly Property FirstDivisor() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return firstDivisorValue
        End Get
    End Property


    Public ReadOnly Property IsPrime() As Boolean
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return isPrimeValue
        End Get
    End Property
End Class
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  You will receive two compiler warnings:

```
warning CS0067: The event 'AsynchronousPatternExample.PrimeNumberCalculator.ProgressChanged' is never
used
warning CS0067: The event 'AsynchronousPatternExample.PrimeNumberCalculator.CalculatePrimeCompleted' is
never used
```

  These warnings will be cleared in the next section.

Defining Private Delegates

The asynchronous aspects of the `PrimeNumberCalculator` component are implemented internally with a special delegate known as a SendOrPostCallback. A SendOrPostCallback represents a callback method that executes on a ThreadPool thread. The callback method must have a signature that takes a single parameter of type Object, which means you will need to pass state among delegates in a wrapper class. For more information, see SendOrPostCallback.

To implement your component's internal asynchronous behavior:

1. Declare and create the SendOrPostCallback delegates in the `PrimeNumberCalculator` class. Create the SendOrPostCallback objects in a utility method called `InitializeDelegates`.

   You will need two delegates: one for reporting progress to the client, and one for reporting completion to the client.

```
private SendOrPostCallback onProgressReportDelegate;
private SendOrPostCallback onCompletedDelegate;
```

```
Private onProgressReportDelegate As SendOrPostCallback
Private onCompletedDelegate As SendOrPostCallback
```

```
protected virtual void InitializeDelegates()
{
    onProgressReportDelegate =
        new SendOrPostCallback(ReportProgress);
    onCompletedDelegate =
        new SendOrPostCallback(CalculateCompleted);
}
```

```
Protected Overridable Sub InitializeDelegates()
    onProgressReportDelegate = _
        New SendOrPostCallback(AddressOf ReportProgress)
    onCompletedDelegate = _
        New SendOrPostCallback(AddressOf CalculateCompleted)
End Sub
```

2. Call the `InitializeDelegates` method in your component's constructor.

```
public PrimeNumberCalculator()
{
    InitializeComponent();

    InitializeDelegates();
}
```

```
Public Sub New()

    InitializeComponent()

    InitializeDelegates()

End Sub
```

3. Declare a delegate in the `PrimeNumberCalculator` class that handles the actual work to be done asynchronously. This delegate wraps the worker method that tests whether a number is prime. The delegate takes an AsyncOperation parameter, which will be used to track the lifetime of the asynchronous operation.

```
private delegate void WorkerEventHandler(
    int numberToCheck,
    AsyncOperation asyncOp);
```

```
Private Delegate Sub WorkerEventHandler( _
ByVal numberToCheck As Integer, _
ByVal asyncOp As AsyncOperation)
```

4. Create a collection for managing lifetimes of pending asynchronous operations. The client needs a way to track operations as they are executed and completed, and this tracking is done by requiring the client to pass a unique token, or task ID, when the client makes the call to the asynchronous method. The `PrimeNumberCalculator` component must keep track of each call by associating the task ID with its corresponding invocation. If the client passes a task ID that is not unique, the `PrimeNumberCalculator` component must raise an exception.

   The `PrimeNumberCalculator` component keeps track of task ID by using a special collection class called a HybridDictionary. In the class definition, create a HybridDictionary called `userTokenToLifetime`.

```
private HybridDictionary userStateToLifetime =
    new HybridDictionary();
```

```
Private userStateToLifetime As New HybridDictionary()
```

Implementing Public Events

Components that implement the Event-based Asynchronous Pattern communicate to clients using events. These events are invoked on the proper thread with the help of the AsyncOperation class.

To raise events to your component's clients:

1. Implement public events for reporting to clients. You will need an event for reporting progress and one for reporting completion.

```
// This method is invoked via the AsyncOperation object,
// so it is guaranteed to be executed on the correct thread.
private void CalculateCompleted(object operationState)
{
    CalculatePrimeCompletedEventArgs e =
        operationState as CalculatePrimeCompletedEventArgs;

    OnCalculatePrimeCompleted(e);
}

// This method is invoked via the AsyncOperation object,
// so it is guaranteed to be executed on the correct thread.
private void ReportProgress(object state)
{
    ProgressChangedEventArgs e =
        state as ProgressChangedEventArgs;

    OnProgressChanged(e);
}

protected void OnCalculatePrimeCompleted(
    CalculatePrimeCompletedEventArgs e)
{
    if (CalculatePrimeCompleted != null)
    {
        CalculatePrimeCompleted(this, e);
    }
}

protected void OnProgressChanged(ProgressChangedEventArgs e)
{
    if (ProgressChanged != null)
    {
        ProgressChanged(e);
    }
}
```

```
' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub CalculateCompleted(ByVal operationState As Object)
    Dim e As CalculatePrimeCompletedEventArgs = operationState

    OnCalculatePrimeCompleted(e)

End Sub


' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub ReportProgress(ByVal state As Object)
    Dim e As ProgressChangedEventArgs = state

    OnProgressChanged(e)

End Sub

Protected Sub OnCalculatePrimeCompleted( _
    ByVal e As CalculatePrimeCompletedEventArgs)

    RaiseEvent CalculatePrimeCompleted(Me, e)

End Sub


Protected Sub OnProgressChanged( _
    ByVal e As ProgressChangedEventArgs)

    RaiseEvent ProgressChanged(e)

End Sub
```

Implementing the Completion Method

The completion delegate is the method that the underlying, free-threaded asynchronous behavior will invoke when the asynchronous operation ends by successful completion, error, or cancellation. This invocation happens on an arbitrary thread.

This method is where the client's task ID is removed from the internal collection of unique client tokens. This method also ends the lifetime of a particular asynchronous operation by calling the PostOperationCompleted method on the corresponding AsyncOperation. This call raises the completion event on the thread that is appropriate for the application model. After the PostOperationCompleted method is called, this instance of AsyncOperation can no longer be used, and any subsequent attempts to use it will throw an exception.

The `CompletionMethod` signature must hold all state necessary to describe the outcome of the asynchronous operation. It holds state for the number that was tested by this particular asynchronous operation, whether the number is prime, and the value of its first divisor if it is not a prime number. It also holds state describing any exception that occurred, and the AsyncOperation corresponding to this particular task.

To complete an asynchronous operation:

- Implement the completion method. It takes six parameters, which it uses to populate a `CalculatePrimeCompletedEventArgs` that is returned to the client through the client's `CalculatePrimeCompletedEventHandler`. It removes the client's task ID token from the internal collection, and it ends the asynchronous operation's lifetime with a call to PostOperationCompleted. The AsyncOperation marshals the call to the thread or context that is appropriate for the application model.

```
// This is the method that the underlying, free-threaded
// asynchronous behavior will invoke.  This will happen on
// an arbitrary thread.
private void CompletionMethod(
    int numberToTest,
    int firstDivisor,
    bool isPrime,
    Exception exception,
    bool canceled,
    AsyncOperation asyncOp )

{
    // If the task was not previously canceled,
    // remove the task from the lifetime collection.
    if (!canceled)
    {
        lock (userStateToLifetime.SyncRoot)
        {
            userStateToLifetime.Remove(asyncOp.UserSuppliedState);
        }
    }

    // Package the results of the operation in a
    // CalculatePrimeCompletedEventArgs.
    CalculatePrimeCompletedEventArgs e =
        new CalculatePrimeCompletedEventArgs(
        numberToTest,
        firstDivisor,
        isPrime,
        exception,
        canceled,
        asyncOp.UserSuppliedState);

    // End the task. The asyncOp object is responsible
    // for marshaling the call.
    asyncOp.PostOperationCompleted(onCompletedDelegate, e);

    // Note that after the call to OperationCompleted,
    // asyncOp is no longer usable, and any attempt to use it
    // will cause an exception to be thrown.
}
```

```
' This is the method that the underlying, free-threaded
' asynchronous behavior will invoke.  This will happen on
'  an arbitrary thread.
Private Sub CompletionMethod( _
    ByVal numberToTest As Integer, _
    ByVal firstDivisor As Integer, _
    ByVal prime As Boolean, _
    ByVal exc As Exception, _
    ByVal canceled As Boolean, _
    ByVal asyncOp As AsyncOperation)

    ' If the task was not previously canceled,
    ' remove the task from the lifetime collection.
    If Not canceled Then
        SyncLock userStateToLifetime.SyncRoot
            userStateToLifetime.Remove(asyncOp.UserSuppliedState)
        End SyncLock
    End If

    ' Package the results of the operation in a
    ' CalculatePrimeCompletedEventArgs.
    Dim e As New CalculatePrimeCompletedEventArgs( _
        numberToTest, _
        firstDivisor, _
        prime, _
        exc, _
        canceled, _
        asyncOp.UserSuppliedState)

    ' End the task. The asyncOp object is responsible
    ' for marshaling the call.
    asyncOp.PostOperationCompleted(onCompletedDelegate, e)

    ' Note that after the call to PostOperationCompleted, asyncOp
    ' is no longer usable, and any attempt to use it will cause.
    ' an exception to be thrown.

End Sub
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  You will receive one compiler warning:

  ```
  warning CS0169: The private field 'AsynchronousPatternExample.PrimeNumberCalculator.workerDelegate' is
  never used
  ```

  This warning will be resolved in the next section.

Implementing the Worker Methods

So far, you have implemented the supporting asynchronous code for the `PrimeNumberCalculator` component. Now you can implement the code that does the actual work. You will implement three methods: `CalculateWorker`, `BuildPrimeNumberList`, and `IsPrime`. Together, `BuildPrimeNumberList` and `IsPrime` comprise a well-known algorithm called the Sieve of Eratosthenes, which determines if a number is prime by finding all the prime numbers up to the square root of the test number. If no divisors are found by that point, the test number is prime.

If this component were written for maximum efficiency, it would remember all the prime numbers discovered by various invocations for different test numbers. It would also check for trivial divisors like 2, 3, and 5. The intent of this example is to demonstrate how time-consuming operations can be executed asynchronously, however, so these optimizations are left as an exercise for you.

The `CalculateWorker` method is wrapped in a delegate and is invoked asynchronously with a call to `BeginInvoke`.

> **NOTE**
>
> Progress reporting is implemented in the `BuildPrimeNumberList` method. On fast computers, `ProgressChanged` events can be raised in rapid succession. The client thread, on which these events are raised, must be able to handle this situation. User interface code may be flooded with messages and unable to keep up, resulting in hanging behavior. For an example user interface that handles this situation, see How to: Implement a Client of the Event-based Asynchronous Pattern.

To execute the prime number calculation asynchronously:

1. Implement the `TaskCanceled` utility method. This checks the task lifetime collection for the given task ID, and returns `true` if the task ID is not found.

```
// Utility method for determining if a
// task has been canceled.
private bool TaskCanceled(object taskId)
{
    return( userStateToLifetime[taskId] == null );
}
```

```
' Utility method for determining if a
' task has been canceled.
Private Function TaskCanceled(ByVal taskId As Object) As Boolean
    Return (userStateToLifetime(taskId) Is Nothing)
End Function
```

2. Implement the `CalculateWorker` method. It takes two parameters: a number to test, and an AsyncOperation.

```
// This method performs the actual prime number computation.
// It is executed on the worker thread.
private void CalculateWorker(
    int numberToTest,
    AsyncOperation asyncOp)
{
    bool isPrime = false;
    int firstDivisor = 1;
    Exception e = null;

    // Check that the task is still active.
    // The operation may have been canceled before
    // the thread was scheduled.
    if (!TaskCanceled(asyncOp.UserSuppliedState))
    {
        try
        {
            // Find all the prime numbers up to
            // the square root of numberToTest.
            ArrayList primes = BuildPrimeNumberList(
                numberToTest,
                asyncOp);

            // Now we have a list of primes less than
            // numberToTest.
            isPrime = IsPrime(
                primes,
                numberToTest,
                out firstDivisor);
        }
        catch (Exception ex)
        {
            e = ex;
        }
    }

    //CalculatePrimeState calcState = new CalculatePrimeState(
    //        numberToTest,
    //        firstDivisor,
    //        isPrime,
    //        e,
    //        TaskCanceled(asyncOp.UserSuppliedState),
    //        asyncOp);

    //this.CompletionMethod(calcState);

    this.CompletionMethod(
        numberToTest,
        firstDivisor,
        isPrime,
        e,
        TaskCanceled(asyncOp.UserSuppliedState),
        asyncOp);

    //completionMethodDelegate(calcState);
}
```

```
' This method performs the actual prime number computation.
' It is executed on the worker thread.
Private Sub CalculateWorker( _
    ByVal numberToTest As Integer, _
    ByVal asyncOp As AsyncOperation)

    Dim prime As Boolean = False
    Dim firstDivisor As Integer = 1
    Dim exc As Exception = Nothing

    ' Check that the task is still active.
    ' The operation may have been canceled before
    ' the thread was scheduled.
    If Not Me.TaskCanceled(asyncOp.UserSuppliedState) Then

        Try
            ' Find all the prime numbers up to the
            ' square root of numberToTest.
            Dim primes As ArrayList = BuildPrimeNumberList( _
                numberToTest, asyncOp)

            ' Now we have a list of primes less than
            'numberToTest.
            prime = IsPrime( _
                primes, _
                numberToTest, _
                firstDivisor)

        Catch ex As Exception
            exc = ex
        End Try

    End If

    Me.CompletionMethod( _
        numberToTest, _
        firstDivisor, _
        prime, _
        exc, _
        TaskCanceled(asyncOp.UserSuppliedState), _
        asyncOp)

End Sub
```

3. Implement `BuildPrimeNumberList`. It takes two parameters: the number to test, and an AsyncOperation. It uses the AsyncOperation to report progress and incremental results. This assures that the client's event handlers are called on the proper thread or context for the application model. When `BuildPrimeNumberList` finds a prime number, it reports this as an incremental result to the client's event handler for the `ProgressChanged` event. This requires a class derived from ProgressChangedEventArgs, called `CalculatePrimeProgressChangedEventArgs`, which has one added property called `LatestPrimeNumber`.

   The `BuildPrimeNumberList` method also periodically calls the `TaskCanceled` method and exits if the method returns `true`.

```
// This method computes the list of prime numbers used by the
// IsPrime method.
private ArrayList BuildPrimeNumberList(
    int numberToTest,
    AsyncOperation asyncOp)
{
    ProgressChangedEventArgs e = null;
    ArrayList primes = new ArrayList();
    int firstDivisor;
    int n = 5;

    // Add the first prime numbers.
    primes.Add(2);
    primes.Add(3);

    // Do the work.
    while (n < numberToTest &&
           !TaskCanceled( asyncOp.UserSuppliedState ) )
    {
        if (IsPrime(primes, n, out firstDivisor))
        {
            // Report to the client that a prime was found.
            e = new CalculatePrimeProgressChangedEventArgs(
                n,
                (int)((float)n / (float)numberToTest * 100),
                asyncOp.UserSuppliedState);

            asyncOp.Post(this.onProgressReportDelegate, e);

            primes.Add(n);

            // Yield the rest of this time slice.
            Thread.Sleep(0);
        }

        // Skip even numbers.
        n += 2;
    }

    return primes;
}
```

```
' This method computes the list of prime numbers used by the
' IsPrime method.
Private Function BuildPrimeNumberList( _
    ByVal numberToTest As Integer, _
    ByVal asyncOp As AsyncOperation) As ArrayList

    Dim e As ProgressChangedEventArgs = Nothing
    Dim primes As New ArrayList
    Dim firstDivisor As Integer
    Dim n As Integer = 5

    ' Add the first prime numbers.
    primes.Add(2)
    primes.Add(3)

    ' Do the work.
    While n < numberToTest And _
        Not Me.TaskCanceled(asyncOp.UserSuppliedState)

        If IsPrime(primes, n, firstDivisor) Then
            ' Report to the client that you found a prime.
            e = New CalculatePrimeProgressChangedEventArgs( _
                n, _
                CSng(n) / CSng(numberToTest) * 100, _
                asyncOp.UserSuppliedState)

            asyncOp.Post(Me.onProgressReportDelegate, e)

            primes.Add(n)

            ' Yield the rest of this time slice.
            Thread.Sleep(0)
        End If

        ' Skip even numbers.
        n += 2

    End While

    Return primes

End Function
```

4. Implement `IsPrime`. It takes three parameters: a list of known prime numbers, the number to test, and an output parameter for the first divisor found. Given the list of prime numbers, it determines if the test number is prime.

```
// This method tests n for primality against the list of
// prime numbers contained in the primes parameter.
private bool IsPrime(
    ArrayList primes,
    int n,
    out int firstDivisor)
{
    bool foundDivisor = false;
    bool exceedsSquareRoot = false;

    int i = 0;
    int divisor = 0;
    firstDivisor = 1;

    // Stop the search if:
    // there are no more primes in the list,
    // there is a divisor of n in the list, or
    // there is a prime that is larger than
    // the square root of n.
    while (
        (i < primes.Count) &&
        !foundDivisor &&
        !exceedsSquareRoot)
    {
        // The divisor variable will be the smallest
        // prime number not yet tried.
        divisor = (int)primes[i++];

        // Determine whether the divisor is greater
        // than the square root of n.
        if (divisor * divisor > n)
        {
            exceedsSquareRoot = true;
        }
        // Determine whether the divisor is a factor of n.
        else if (n % divisor == 0)
        {
            firstDivisor = divisor;
            foundDivisor = true;
        }
    }

    return !foundDivisor;
}
```

```
' This method tests n for primality against the list of
' prime numbers contained in the primes parameter.
Private Function IsPrime( _
    ByVal primes As ArrayList, _
    ByVal n As Integer, _
    ByRef firstDivisor As Integer) As Boolean

    Dim foundDivisor As Boolean = False
    Dim exceedsSquareRoot As Boolean = False

    Dim i As Integer = 0
    Dim divisor As Integer = 0
    firstDivisor = 1

    ' Stop the search if:
    ' there are no more primes in the list,
    ' there is a divisor of n in the list, or
    ' there is a prime that is larger than
    ' the square root of n.
    While i < primes.Count AndAlso _
        Not foundDivisor AndAlso _
        Not exceedsSquareRoot

        ' The divisor variable will be the smallest prime number
        ' not yet tried.
        divisor = primes(i)
        i = i + 1

        ' Determine whether the divisor is greater than the
        ' square root of n.
        If divisor * divisor > n Then
            exceedsSquareRoot = True
            ' Determine whether the divisor is a factor of n.
        ElseIf n Mod divisor = 0 Then
            firstDivisor = divisor
            foundDivisor = True
        End If
    End While

    Return Not foundDivisor

End Function
```

5. Derive `CalculatePrimeProgressChangedEventArgs` from ProgressChangedEventArgs. This class is necessary for reporting incremental results to the client's event handler for the `ProgressChanged` event. It has one added property called `LatestPrimeNumber`.

```
public class CalculatePrimeProgressChangedEventArgs :
        ProgressChangedEventArgs
{
    private int latestPrimeNumberValue = 1;

    public CalculatePrimeProgressChangedEventArgs(
        int latestPrime,
        int progressPercentage,
        object userToken) : base( progressPercentage, userToken )
    {
        this.latestPrimeNumberValue = latestPrime;
    }

    public int LatestPrimeNumber
    {
        get
        {
            return latestPrimeNumberValue;
        }
    }
}
```

```
Public Class CalculatePrimeProgressChangedEventArgs
    Inherits ProgressChangedEventArgs
    Private latestPrimeNumberValue As Integer = 1


    Public Sub New( _
        ByVal latestPrime As Integer, _
        ByVal progressPercentage As Integer, _
        ByVal UserState As Object)

        MyBase.New(progressPercentage, UserState)
        Me.latestPrimeNumberValue = latestPrime

    End Sub

    Public ReadOnly Property LatestPrimeNumber() As Integer
        Get
            Return latestPrimeNumberValue
        End Get
    End Property
End Class
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  All that remains to be written are the methods to start and cancel asynchronous operations, `CalculatePrimeAsync` and `CancelAsync`.

Implementing the Start and Cancel Methods

You start the worker method on its own thread by calling `BeginInvoke` on the delegate that wraps it. To manage the lifetime of a particular asynchronous operation, you call the CreateOperation method on the AsyncOperationManager helper class. This returns an AsyncOperation, which marshals calls on the client's event handlers to the proper thread or context.

You cancel a particular pending operation by calling PostOperationCompleted on its corresponding AsyncOperation. This ends that operation, and any subsequent calls to its AsyncOperation will throw an exception.

To implement Start and Cancel functionality:

1. Implement the `CalculatePrimeAsync` method. Make sure the client-supplied token (task ID) is unique with respect to all the tokens representing currently pending tasks. If the client passes in a non-unique token, `CalculatePrimeAsync` raises an exception. Otherwise, the token is added to the task ID collection.

```
// This method starts an asynchronous calculation.
// First, it checks the supplied task ID for uniqueness.
// If taskId is unique, it creates a new WorkerEventHandler
// and calls its BeginInvoke method to start the calculation.
public virtual void CalculatePrimeAsync(
    int numberToTest,
    object taskId)
{
    // Create an AsyncOperation for taskId.
    AsyncOperation asyncOp =
        AsyncOperationManager.CreateOperation(taskId);

    // Multiple threads will access the task dictionary,
    // so it must be locked to serialize access.
    lock (userStateToLifetime.SyncRoot)
    {
        if (userStateToLifetime.Contains(taskId))
        {
            throw new ArgumentException(
                "Task ID parameter must be unique",
                "taskId");
        }

        userStateToLifetime[taskId] = asyncOp;
    }

    // Start the asynchronous operation.
    WorkerEventHandler workerDelegate = new WorkerEventHandler(CalculateWorker);
    workerDelegate.BeginInvoke(
        numberToTest,
        asyncOp,
        null,
        null);
}
```

```
' This method starts an asynchronous calculation.
' First, it checks the supplied task ID for uniqueness.
' If taskId is unique, it creates a new WorkerEventHandler
' and calls its BeginInvoke method to start the calculation.
Public Overridable Sub CalculatePrimeAsync( _
    ByVal numberToTest As Integer, _
    ByVal taskId As Object)

    ' Create an AsyncOperation for taskId.
    Dim asyncOp As AsyncOperation = _
        AsyncOperationManager.CreateOperation(taskId)

    ' Multiple threads will access the task dictionary,
    ' so it must be locked to serialize access.
    SyncLock userStateToLifetime.SyncRoot
        If userStateToLifetime.Contains(taskId) Then
            Throw New ArgumentException( _
                "Task ID parameter must be unique", _
                "taskId")
        End If

        userStateToLifetime(taskId) = asyncOp
    End SyncLock

    ' Start the asynchronous operation.
    Dim workerDelegate As New WorkerEventHandler( _
        AddressOf CalculateWorker)

    workerDelegate.BeginInvoke( _
        numberToTest, _
        asyncOp, _
        Nothing, _
        Nothing)

End Sub
```

2. Implement the `CancelAsync` method. If the `taskId` parameter exists in the token collection, it is removed. This prevents canceled tasks that have not started from running. If the task is running, the `BuildPrimeNumberList` method exits when it detects that the task ID has been removed from the lifetime collection.

```
// This method cancels a pending asynchronous operation.
public void CancelAsync(object taskId)
{
    AsyncOperation asyncOp = userStateToLifetime[taskId] as AsyncOperation;
    if (asyncOp != null)
    {
        lock (userStateToLifetime.SyncRoot)
        {
            userStateToLifetime.Remove(taskId);
        }
    }
}
```

```
' This method cancels a pending asynchronous operation.
Public Sub CancelAsync(ByVal taskId As Object)

    Dim obj As Object = userStateToLifetime(taskId)
    If (obj IsNot Nothing) Then

        SyncLock userStateToLifetime.SyncRoot

            userStateToLifetime.Remove(taskId)

        End SyncLock

    End If

End Sub
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

The `PrimeNumberCalculator` component is now complete and ready to use.

For an example client that uses the `PrimeNumberCalculator` component, see How to: Implement a Client of the Event-based Asynchronous Pattern.

Next Steps

You can fill out this example by writing `CalculatePrime`, the synchronous equivalent of `CalculatePrimeAsync` method. This will make the `PrimeNumberCalculator` component fully compliant with the Event-based Asynchronous Pattern.

You can improve this example by retaining the list of all the prime numbers discovered by various invocations for different test numbers. Using this approach, each task will benefit from the work done by previous tasks. Be careful to protect this list with `lock` regions, so access to the list by different threads is serialized.

You can also improve this example by testing for trivial divisors, like 2, 3, and 5.

See Also

How to: Run an Operation in the Background
Event-based Asynchronous Pattern Overview
NOT IN BUILD: Multithreading in Visual Basic
How to: Implement a Component That Supports the Event-based Asynchronous Pattern
Multithreaded Programming with the Event-based Asynchronous Pattern

Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern

11/21/2017 • 21 min to read • Edit Online

If you are writing a class with some operations that may incur noticeable delays, consider giving it asynchronous functionality by implementing the Event-based Asynchronous Pattern Overview.

This walkthrough illustrates how to create a component that implements the Event-based Asynchronous Pattern. It is implemented using helper classes from the System.ComponentModel namespace, which ensures that the component works correctly under any application model, including ASP.NET, Console applications and Windows Forms applications. This component is also designable with a PropertyGrid control and your own custom designers.

When you are through, you will have an application that computes prime numbers asynchronously. Your application will have a main user interface (UI) thread and a thread for each prime number calculation. Although testing whether a large number is prime can take a noticeable amount of time, the main UI thread will not be interrupted by this delay, and the form will be responsive during the calculations. You will be able to run as many calculations as you like concurrently and selectively cancel pending calculations.

Tasks illustrated in this walkthrough include:

- Creating the Component
- Defining Public Asynchronous Events and Delegates
- Defining Private Delegates
- Implementing Public Events
- Implementing the Completion Method
- Implementing the Worker Methods
- Implementing Start and Cancel Methods

To copy the code in this topic as a single listing, see How to: Implement a Component That Supports the Event-based Asynchronous Pattern.

Creating the Component

The first step is to create the component that will implement the Event-based Asynchronous Pattern.

To create the component

- Create a class called `PrimeNumberCalculator` that inherits from Component.

Defining Public Asynchronous Events and Delegates

Your component communicates to clients using events. The *MethodName* `Completed` event alerts clients to the completion of an asynchronous task, and the *MethodName* `ProgressChanged` event informs clients of the progress of an asynchronous task.

To define asynchronous events for clients of your component:

1. Import the System.Threading and System.Collections.Specialized namespaces at the top of your file.

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Globalization;
using System.Threading;
using System.Windows.Forms;
```

```
Imports System
Imports System.Collections
Imports System.Collections.Specialized
Imports System.ComponentModel
Imports System.Drawing
Imports System.Globalization
Imports System.Threading
Imports System.Windows.Forms
```

2. Before the `PrimeNumberCalculator` class definition, declare delegates for progress and completion events.

```
public delegate void ProgressChangedEventHandler(
    ProgressChangedEventArgs e);

public delegate void CalculatePrimeCompletedEventHandler(
    object sender,
    CalculatePrimeCompletedEventArgs e);
```

```
Public Delegate Sub ProgressChangedEventHandler( _
    ByVal e As ProgressChangedEventArgs)

Public Delegate Sub CalculatePrimeCompletedEventHandler( _
    ByVal sender As Object, _
    ByVal e As CalculatePrimeCompletedEventArgs)
```

3. In the `PrimeNumberCalculator` class definition, declare events for reporting progress and completion to clients.

```
public event ProgressChangedEventHandler ProgressChanged;
public event CalculatePrimeCompletedEventHandler CalculatePrimeCompleted;
```

```
Public Event ProgressChanged _
    As ProgressChangedEventHandler
Public Event CalculatePrimeCompleted _
    As CalculatePrimeCompletedEventHandler
```

4. After the `PrimeNumberCalculator` class definition, derive the `CalculatePrimeCompletedEventArgs` class for reporting the outcome of each calculation to the client's event handler for the `CalculatePrimeCompleted` .event. In addition to the `AsyncCompletedEventArgs` properties, this class enables the client to determine what number was tested, whether it is prime, and what the first divisor is if it is not prime.

```
public class CalculatePrimeCompletedEventArgs :
    AsyncCompletedEventArgs
{
    private int numberToTestValue = 0;
    private int firstDivisorValue = 1;
    private bool isPrimeValue;

    public CalculatePrimeCompletedEventArgs(
        int numberToTest,
        int firstDivisor,
        bool isPrime,
        Exception e,
        bool canceled,
        object state) : base(e, canceled, state)
    {
        this.numberToTestValue = numberToTest;
        this.firstDivisorValue = firstDivisor;
        this.isPrimeValue = isPrime;
    }

    public int NumberToTest
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return numberToTestValue;
        }
    }

    public int FirstDivisor
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return firstDivisorValue;
        }
    }

    public bool IsPrime
    {
        get
        {
            // Raise an exception if the operation failed or
            // was canceled.
            RaiseExceptionIfNecessary();

            // If the operation was successful, return the
            // property value.
            return isPrimeValue;
        }
    }
}
```

```
Public Class CalculatePrimeCompletedEventArgs
    Inherits AsyncCompletedEventArgs
    Private numberToTestValue As Integer = 0
    Private firstDivisorValue As Integer = 1
    Private isPrimeValue As Boolean


    Public Sub New( _
    ByVal numberToTest As Integer, _
    ByVal firstDivisor As Integer, _
    ByVal isPrime As Boolean, _
    ByVal e As Exception, _
    ByVal canceled As Boolean, _
    ByVal state As Object)

        MyBase.New(e, canceled, state)
        Me.numberToTestValue = numberToTest
        Me.firstDivisorValue = firstDivisor
        Me.isPrimeValue = isPrime

    End Sub


    Public ReadOnly Property NumberToTest() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return numberToTestValue
        End Get
    End Property


    Public ReadOnly Property FirstDivisor() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return firstDivisorValue
        End Get
    End Property


    Public ReadOnly Property IsPrime() As Boolean
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return isPrimeValue
        End Get
    End Property
End Class
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  You will receive two compiler warnings:

```
warning CS0067: The event 'AsynchronousPatternExample.PrimeNumberCalculator.ProgressChanged' is never
used
warning CS0067: The event 'AsynchronousPatternExample.PrimeNumberCalculator.CalculatePrimeCompleted'
is never used
```

  These warnings will be cleared in the next section.

Defining Private Delegates

The asynchronous aspects of the `PrimeNumberCalculator` component are implemented internally with a special delegate known as a SendOrPostCallback. A SendOrPostCallback represents a callback method that executes on a ThreadPool thread. The callback method must have a signature that takes a single parameter of type Object, which means you will need to pass state among delegates in a wrapper class. For more information, see SendOrPostCallback.

To implement your component's internal asynchronous behavior:

1. Declare and create the SendOrPostCallback delegates in the `PrimeNumberCalculator` class. Create the SendOrPostCallback objects in a utility method called `InitializeDelegates`.

   You will need two delegates: one for reporting progress to the client, and one for reporting completion to the client.

```
private SendOrPostCallback onProgressReportDelegate;
private SendOrPostCallback onCompletedDelegate;
```

```
Private onProgressReportDelegate As SendOrPostCallback
Private onCompletedDelegate As SendOrPostCallback
```

```
protected virtual void InitializeDelegates()
{
    onProgressReportDelegate =
        new SendOrPostCallback(ReportProgress);
    onCompletedDelegate =
        new SendOrPostCallback(CalculateCompleted);
}
```

```
Protected Overridable Sub InitializeDelegates()
    onProgressReportDelegate = _
        New SendOrPostCallback(AddressOf ReportProgress)
    onCompletedDelegate = _
        New SendOrPostCallback(AddressOf CalculateCompleted)
End Sub
```

2. Call the `InitializeDelegates` method in your component's constructor.

```
public PrimeNumberCalculator()
{
    InitializeComponent();

    InitializeDelegates();
}
```

```
Public Sub New()

    InitializeComponent()

    InitializeDelegates()

End Sub
```

3. Declare a delegate in the `PrimeNumberCalculator` class that handles the actual work to be done asynchronously. This delegate wraps the worker method that tests whether a number is prime. The delegate takes an AsyncOperation parameter, which will be used to track the lifetime of the asynchronous operation.

```
private delegate void WorkerEventHandler(
    int numberToCheck,
    AsyncOperation asyncOp);
```

```
Private Delegate Sub WorkerEventHandler( _
ByVal numberToCheck As Integer, _
ByVal asyncOp As AsyncOperation)
```

4. Create a collection for managing lifetimes of pending asynchronous operations. The client needs a way to track operations as they are executed and completed, and this tracking is done by requiring the client to pass a unique token, or task ID, when the client makes the call to the asynchronous method. The `PrimeNumberCalculator` component must keep track of each call by associating the task ID with its corresponding invocation. If the client passes a task ID that is not unique, the `PrimeNumberCalculator` component must raise an exception.

   The `PrimeNumberCalculator` component keeps track of task ID by using a special collection class called a HybridDictionary. In the class definition, create a HybridDictionary called `userTokenToLifetime`.

```
private HybridDictionary userStateToLifetime =
    new HybridDictionary();
```

```
Private userStateToLifetime As New HybridDictionary()
```

Implementing Public Events

Components that implement the Event-based Asynchronous Pattern communicate to clients using events. These events are invoked on the proper thread with the help of the AsyncOperation class.

To raise events to your component's clients:

1. Implement public events for reporting to clients. You will need an event for reporting progress and one for reporting completion.

```
// This method is invoked via the AsyncOperation object,
// so it is guaranteed to be executed on the correct thread.
private void CalculateCompleted(object operationState)
{
    CalculatePrimeCompletedEventArgs e =
        operationState as CalculatePrimeCompletedEventArgs;

    OnCalculatePrimeCompleted(e);
}

// This method is invoked via the AsyncOperation object,
// so it is guaranteed to be executed on the correct thread.
private void ReportProgress(object state)
{
    ProgressChangedEventArgs e =
        state as ProgressChangedEventArgs;

    OnProgressChanged(e);
}

protected void OnCalculatePrimeCompleted(
    CalculatePrimeCompletedEventArgs e)
{
    if (CalculatePrimeCompleted != null)
    {
        CalculatePrimeCompleted(this, e);
    }
}

protected void OnProgressChanged(ProgressChangedEventArgs e)
{
    if (ProgressChanged != null)
    {
        ProgressChanged(e);
    }
}
```

```
' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub CalculateCompleted(ByVal operationState As Object)
    Dim e As CalculatePrimeCompletedEventArgs = operationState

    OnCalculatePrimeCompleted(e)

End Sub


' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub ReportProgress(ByVal state As Object)
    Dim e As ProgressChangedEventArgs = state

    OnProgressChanged(e)

End Sub

Protected Sub OnCalculatePrimeCompleted( _
    ByVal e As CalculatePrimeCompletedEventArgs)

    RaiseEvent CalculatePrimeCompleted(Me, e)

End Sub


Protected Sub OnProgressChanged( _
    ByVal e As ProgressChangedEventArgs)

    RaiseEvent ProgressChanged(e)

End Sub
```

Implementing the Completion Method

The completion delegate is the method that the underlying, free-threaded asynchronous behavior will invoke when the asynchronous operation ends by successful completion, error, or cancellation. This invocation happens on an arbitrary thread.

This method is where the client's task ID is removed from the internal collection of unique client tokens. This method also ends the lifetime of a particular asynchronous operation by calling the PostOperationCompleted method on the corresponding AsyncOperation. This call raises the completion event on the thread that is appropriate for the application model. After the PostOperationCompleted method is called, this instance of AsyncOperation can no longer be used, and any subsequent attempts to use it will throw an exception.

The `CompletionMethod` signature must hold all state necessary to describe the outcome of the asynchronous operation. It holds state for the number that was tested by this particular asynchronous operation, whether the number is prime, and the value of its first divisor if it is not a prime number. It also holds state describing any exception that occurred, and the AsyncOperation corresponding to this particular task.

To complete an asynchronous operation:

- Implement the completion method. It takes six parameters, which it uses to populate a `CalculatePrimeCompletedEventArgs` that is returned to the client through the client's `CalculatePrimeCompletedEventHandler`. It removes the client's task ID token from the internal collection, and it ends the asynchronous operation's lifetime with a call to PostOperationCompleted. The AsyncOperation marshals the call to the thread or context that is appropriate for the application model.

```
// This is the method that the underlying, free-threaded
// asynchronous behavior will invoke.  This will happen on
// an arbitrary thread.
private void CompletionMethod(
    int numberToTest,
    int firstDivisor,
    bool isPrime,
    Exception exception,
    bool canceled,
    AsyncOperation asyncOp )

{
    // If the task was not previously canceled,
    // remove the task from the lifetime collection.
    if (!canceled)
    {
        lock (userStateToLifetime.SyncRoot)
        {
            userStateToLifetime.Remove(asyncOp.UserSuppliedState);
        }
    }

    // Package the results of the operation in a
    // CalculatePrimeCompletedEventArgs.
    CalculatePrimeCompletedEventArgs e =
        new CalculatePrimeCompletedEventArgs(
        numberToTest,
        firstDivisor,
        isPrime,
        exception,
        canceled,
        asyncOp.UserSuppliedState);

    // End the task. The asyncOp object is responsible
    // for marshaling the call.
    asyncOp.PostOperationCompleted(onCompletedDelegate, e);

    // Note that after the call to OperationCompleted,
    // asyncOp is no longer usable, and any attempt to use it
    // will cause an exception to be thrown.
}
```

```
' This is the method that the underlying, free-threaded
' asynchronous behavior will invoke.  This will happen on
'  an arbitrary thread.
Private Sub CompletionMethod( _
    ByVal numberToTest As Integer, _
    ByVal firstDivisor As Integer, _
    ByVal prime As Boolean, _
    ByVal exc As Exception, _
    ByVal canceled As Boolean, _
    ByVal asyncOp As AsyncOperation)

    ' If the task was not previously canceled,
    ' remove the task from the lifetime collection.
    If Not canceled Then
        SyncLock userStateToLifetime.SyncRoot
            userStateToLifetime.Remove(asyncOp.UserSuppliedState)
        End SyncLock
    End If

    ' Package the results of the operation in a
    ' CalculatePrimeCompletedEventArgs.
    Dim e As New CalculatePrimeCompletedEventArgs( _
        numberToTest, _
        firstDivisor, _
        prime, _
        exc, _
        canceled, _
        asyncOp.UserSuppliedState)

    ' End the task. The asyncOp object is responsible
    ' for marshaling the call.
    asyncOp.PostOperationCompleted(onCompletedDelegate, e)

    ' Note that after the call to PostOperationCompleted, asyncOp
    ' is no longer usable, and any attempt to use it will cause.
    ' an exception to be thrown.

End Sub
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  You will receive one compiler warning:

  ```
  warning CS0169: The private field 'AsynchronousPatternExample.PrimeNumberCalculator.workerDelegate'
  is never used
  ```

  This warning will be resolved in the next section.

Implementing the Worker Methods

So far, you have implemented the supporting asynchronous code for the `PrimeNumberCalculator` component. Now you can implement the code that does the actual work. You will implement three methods: `CalculateWorker`, `BuildPrimeNumberList`, and `IsPrime`. Together, `BuildPrimeNumberList` and `IsPrime` comprise a well-known algorithm called the Sieve of Eratosthenes, which determines if a number is prime by finding all the prime numbers up to the square root of the test number. If no divisors are found by that point, the test number is prime.

If this component were written for maximum efficiency, it would remember all the prime numbers discovered by various invocations for different test numbers. It would also check for trivial divisors like 2, 3, and 5. The intent of this example is to demonstrate how time-consuming operations can be executed asynchronously, however, so these optimizations are left as an exercise for you.

The `CalculateWorker` method is wrapped in a delegate and is invoked asynchronously with a call to `BeginInvoke` .

> **NOTE**
>
> Progress reporting is implemented in the `BuildPrimeNumberList` method. On fast computers, `ProgressChanged` events can be raised in rapid succession. The client thread, on which these events are raised, must be able to handle this situation. User interface code may be flooded with messages and unable to keep up, resulting in hanging behavior. For an example user interface that handles this situation, see How to: Implement a Client of the Event-based Asynchronous Pattern.

To execute the prime number calculation asynchronously:

1. Implement the `TaskCanceled` utility method. This checks the task lifetime collection for the given task ID, and returns `true` if the task ID is not found.

```
// Utility method for determining if a
// task has been canceled.
private bool TaskCanceled(object taskId)
{
    return( userStateToLifetime[taskId] == null );
}
```

```
' Utility method for determining if a
' task has been canceled.
Private Function TaskCanceled(ByVal taskId As Object) As Boolean
    Return (userStateToLifetime(taskId) Is Nothing)
End Function
```

2. Implement the `CalculateWorker` method. It takes two parameters: a number to test, and an AsyncOperation.

```
// This method performs the actual prime number computation.
// It is executed on the worker thread.
private void CalculateWorker(
    int numberToTest,
    AsyncOperation asyncOp)
{
    bool isPrime = false;
    int firstDivisor = 1;
    Exception e = null;

    // Check that the task is still active.
    // The operation may have been canceled before
    // the thread was scheduled.
    if (!TaskCanceled(asyncOp.UserSuppliedState))
    {
        try
        {
            // Find all the prime numbers up to
            // the square root of numberToTest.
            ArrayList primes = BuildPrimeNumberList(
                numberToTest,
                asyncOp);

            // Now we have a list of primes less than
            // numberToTest.
            isPrime = IsPrime(
                primes,
                numberToTest,
                out firstDivisor);
        }
        catch (Exception ex)
        {
            e = ex;
        }
    }

    //CalculatePrimeState calcState = new CalculatePrimeState(
    //        numberToTest,
    //        firstDivisor,
    //        isPrime,
    //        e,
    //        TaskCanceled(asyncOp.UserSuppliedState),
    //        asyncOp);

    //this.CompletionMethod(calcState);

    this.CompletionMethod(
        numberToTest,
        firstDivisor,
        isPrime,
        e,
        TaskCanceled(asyncOp.UserSuppliedState),
        asyncOp);

    //completionMethodDelegate(calcState);
}
```

```
' This method performs the actual prime number computation.
' It is executed on the worker thread.
Private Sub CalculateWorker( _
    ByVal numberToTest As Integer, _
    ByVal asyncOp As AsyncOperation)

    Dim prime As Boolean = False
    Dim firstDivisor As Integer = 1
    Dim exc As Exception = Nothing

    ' Check that the task is still active.
    ' The operation may have been canceled before
    ' the thread was scheduled.
    If Not Me.TaskCanceled(asyncOp.UserSuppliedState) Then

        Try
            ' Find all the prime numbers up to the
            ' square root of numberToTest.
            Dim primes As ArrayList = BuildPrimeNumberList( _
                numberToTest, asyncOp)

            ' Now we have a list of primes less than
            'numberToTest.
            prime = IsPrime( _
                primes, _
                numberToTest, _
                firstDivisor)

        Catch ex As Exception
            exc = ex
        End Try

    End If

    Me.CompletionMethod( _
        numberToTest, _
        firstDivisor, _
        prime, _
        exc, _
        TaskCanceled(asyncOp.UserSuppliedState), _
        asyncOp)

End Sub
```

3. Implement `BuildPrimeNumberList`. It takes two parameters: the number to test, and an AsyncOperation. It uses the AsyncOperation to report progress and incremental results. This assures that the client's event handlers are called on the proper thread or context for the application model. When `BuildPrimeNumberList` finds a prime number, it reports this as an incremental result to the client's event handler for the `ProgressChanged` event. This requires a class derived from ProgressChangedEventArgs, called `CalculatePrimeProgressChangedEventArgs`, which has one added property called `LatestPrimeNumber`.

   The `BuildPrimeNumberList` method also periodically calls the `TaskCanceled` method and exits if the method returns `true`.

```
// This method computes the list of prime numbers used by the
// IsPrime method.
private ArrayList BuildPrimeNumberList(
    int numberToTest,
    AsyncOperation asyncOp)
{
    ProgressChangedEventArgs e = null;
    ArrayList primes = new ArrayList();
    int firstDivisor;
    int n = 5;

    // Add the first prime numbers.
    primes.Add(2);
    primes.Add(3);

    // Do the work.
    while (n < numberToTest &&
           !TaskCanceled( asyncOp.UserSuppliedState ) )
    {
        if (IsPrime(primes, n, out firstDivisor))
        {
            // Report to the client that a prime was found.
            e = new CalculatePrimeProgressChangedEventArgs(
                n,
                (int)((float)n / (float)numberToTest * 100),
                asyncOp.UserSuppliedState);

            asyncOp.Post(this.onProgressReportDelegate, e);

            primes.Add(n);

            // Yield the rest of this time slice.
            Thread.Sleep(0);
        }

        // Skip even numbers.
        n += 2;
    }

    return primes;
}
```

```
' This method computes the list of prime numbers used by the
' IsPrime method.
Private Function BuildPrimeNumberList( _
    ByVal numberToTest As Integer, _
    ByVal asyncOp As AsyncOperation) As ArrayList

    Dim e As ProgressChangedEventArgs = Nothing
    Dim primes As New ArrayList
    Dim firstDivisor As Integer
    Dim n As Integer = 5

    ' Add the first prime numbers.
    primes.Add(2)
    primes.Add(3)

    ' Do the work.
    While n < numberToTest And _
        Not Me.TaskCanceled(asyncOp.UserSuppliedState)

        If IsPrime(primes, n, firstDivisor) Then
            ' Report to the client that you found a prime.
            e = New CalculatePrimeProgressChangedEventArgs( _
                n, _
                CSng(n) / CSng(numberToTest) * 100, _
                asyncOp.UserSuppliedState)

            asyncOp.Post(Me.onProgressReportDelegate, e)

            primes.Add(n)

            ' Yield the rest of this time slice.
            Thread.Sleep(0)
        End If

        ' Skip even numbers.
        n += 2

    End While

    Return primes

End Function
```

4. Implement `IsPrime`. It takes three parameters: a list of known prime numbers, the number to test, and an output parameter for the first divisor found. Given the list of prime numbers, it determines if the test number is prime.

```
// This method tests n for primality against the list of
// prime numbers contained in the primes parameter.
private bool IsPrime(
    ArrayList primes,
    int n,
    out int firstDivisor)
{
    bool foundDivisor = false;
    bool exceedsSquareRoot = false;

    int i = 0;
    int divisor = 0;
    firstDivisor = 1;

    // Stop the search if:
    // there are no more primes in the list,
    // there is a divisor of n in the list, or
    // there is a prime that is larger than
    // the square root of n.
    while (
        (i < primes.Count) &&
        !foundDivisor &&
        !exceedsSquareRoot)
    {
        // The divisor variable will be the smallest
        // prime number not yet tried.
        divisor = (int)primes[i++];

        // Determine whether the divisor is greater
        // than the square root of n.
        if (divisor * divisor > n)
        {
            exceedsSquareRoot = true;
        }
        // Determine whether the divisor is a factor of n.
        else if (n % divisor == 0)
        {
            firstDivisor = divisor;
            foundDivisor = true;
        }
    }

    return !foundDivisor;
}
```

```
' This method tests n for primality against the list of
' prime numbers contained in the primes parameter.
Private Function IsPrime( _
    ByVal primes As ArrayList, _
    ByVal n As Integer, _
    ByRef firstDivisor As Integer) As Boolean

    Dim foundDivisor As Boolean = False
    Dim exceedsSquareRoot As Boolean = False

    Dim i As Integer = 0
    Dim divisor As Integer = 0
    firstDivisor = 1

    ' Stop the search if:
    ' there are no more primes in the list,
    ' there is a divisor of n in the list, or
    ' there is a prime that is larger than
    ' the square root of n.
    While i < primes.Count AndAlso _
        Not foundDivisor AndAlso _
        Not exceedsSquareRoot

        ' The divisor variable will be the smallest prime number
        ' not yet tried.
        divisor = primes(i)
        i = i + 1

        ' Determine whether the divisor is greater than the
        ' square root of n.
        If divisor * divisor > n Then
            exceedsSquareRoot = True
            ' Determine whether the divisor is a factor of n.
        ElseIf n Mod divisor = 0 Then
            firstDivisor = divisor
            foundDivisor = True
        End If
    End While

    Return Not foundDivisor

End Function
```

5. Derive `CalculatePrimeProgressChangedEventArgs` from ProgressChangedEventArgs. This class is necessary for reporting incremental results to the client's event handler for the `ProgressChanged` event. It has one added property called `LatestPrimeNumber`.

```
public class CalculatePrimeProgressChangedEventArgs :
        ProgressChangedEventArgs
{
    private int latestPrimeNumberValue = 1;

    public CalculatePrimeProgressChangedEventArgs(
        int latestPrime,
        int progressPercentage,
        object userToken) : base( progressPercentage, userToken )
    {
        this.latestPrimeNumberValue = latestPrime;
    }

    public int LatestPrimeNumber
    {
        get
        {
            return latestPrimeNumberValue;
        }
    }
}
```

```
Public Class CalculatePrimeProgressChangedEventArgs
    Inherits ProgressChangedEventArgs
    Private latestPrimeNumberValue As Integer = 1


    Public Sub New( _
        ByVal latestPrime As Integer, _
        ByVal progressPercentage As Integer, _
        ByVal UserState As Object)

        MyBase.New(progressPercentage, UserState)
        Me.latestPrimeNumberValue = latestPrime

    End Sub

    Public ReadOnly Property LatestPrimeNumber() As Integer
        Get
            Return latestPrimeNumberValue
        End Get
    End Property
End Class
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

  All that remains to be written are the methods to start and cancel asynchronous operations, `CalculatePrimeAsync` and `CancelAsync`.

Implementing the Start and Cancel Methods

You start the worker method on its own thread by calling `BeginInvoke` on the delegate that wraps it. To manage the lifetime of a particular asynchronous operation, you call the CreateOperation method on the AsyncOperationManager helper class. This returns an AsyncOperation, which marshals calls on the client's event handlers to the proper thread or context.

You cancel a particular pending operation by calling PostOperationCompleted on its corresponding AsyncOperation. This ends that operation, and any subsequent calls to its AsyncOperation will throw an exception.

To implement Start and Cancel functionality:

1. Implement the `CalculatePrimeAsync` method. Make sure the client-supplied token (task ID) is unique with respect to all the tokens representing currently pending tasks. If the client passes in a non-unique token, `CalculatePrimeAsync` raises an exception. Otherwise, the token is added to the task ID collection.

```
// This method starts an asynchronous calculation.
// First, it checks the supplied task ID for uniqueness.
// If taskId is unique, it creates a new WorkerEventHandler
// and calls its BeginInvoke method to start the calculation.
public virtual void CalculatePrimeAsync(
    int numberToTest,
    object taskId)
{
    // Create an AsyncOperation for taskId.
    AsyncOperation asyncOp =
        AsyncOperationManager.CreateOperation(taskId);

    // Multiple threads will access the task dictionary,
    // so it must be locked to serialize access.
    lock (userStateToLifetime.SyncRoot)
    {
        if (userStateToLifetime.Contains(taskId))
        {
            throw new ArgumentException(
                "Task ID parameter must be unique",
                "taskId");
        }

        userStateToLifetime[taskId] = asyncOp;
    }

    // Start the asynchronous operation.
    WorkerEventHandler workerDelegate = new WorkerEventHandler(CalculateWorker);
    workerDelegate.BeginInvoke(
        numberToTest,
        asyncOp,
        null,
        null);
}
```

```
' This method starts an asynchronous calculation.
' First, it checks the supplied task ID for uniqueness.
' If taskId is unique, it creates a new WorkerEventHandler
' and calls its BeginInvoke method to start the calculation.
Public Overridable Sub CalculatePrimeAsync( _
    ByVal numberToTest As Integer, _
    ByVal taskId As Object)

    ' Create an AsyncOperation for taskId.
    Dim asyncOp As AsyncOperation = _
        AsyncOperationManager.CreateOperation(taskId)

    ' Multiple threads will access the task dictionary,
    ' so it must be locked to serialize access.
    SyncLock userStateToLifetime.SyncRoot
        If userStateToLifetime.Contains(taskId) Then
            Throw New ArgumentException( _
                "Task ID parameter must be unique", _
                "taskId")
        End If

        userStateToLifetime(taskId) = asyncOp
    End SyncLock

    ' Start the asynchronous operation.
    Dim workerDelegate As New WorkerEventHandler( _
        AddressOf CalculateWorker)

    workerDelegate.BeginInvoke( _
        numberToTest, _
        asyncOp, _
        Nothing, _
        Nothing)

End Sub
```

2. Implement the `CancelAsync` method. If the `taskId` parameter exists in the token collection, it is removed. This prevents canceled tasks that have not started from running. If the task is running, the `BuildPrimeNumberList` method exits when it detects that the task ID has been removed from the lifetime collection.

```
// This method cancels a pending asynchronous operation.
public void CancelAsync(object taskId)
{
    AsyncOperation asyncOp = userStateToLifetime[taskId] as AsyncOperation;
    if (asyncOp != null)
    {
        lock (userStateToLifetime.SyncRoot)
        {
            userStateToLifetime.Remove(taskId);
        }
    }
}
```

```
' This method cancels a pending asynchronous operation.
Public Sub CancelAsync(ByVal taskId As Object)

    Dim obj As Object = userStateToLifetime(taskId)
    If (obj IsNot Nothing) Then

        SyncLock userStateToLifetime.SyncRoot

            userStateToLifetime.Remove(taskId)

        End SyncLock

    End If

End Sub
```

Checkpoint

At this point, you can build the component.

To test your component

- Compile the component.

The `PrimeNumberCalculator` component is now complete and ready to use.

For an example client that uses the `PrimeNumberCalculator` component, see How to: Implement a Client of the Event-based Asynchronous Pattern.

Next Steps

You can fill out this example by writing `CalculatePrime`, the synchronous equivalent of `CalculatePrimeAsync` method. This will make the `PrimeNumberCalculator` component fully compliant with the Event-based Asynchronous Pattern.

You can improve this example by retaining the list of all the prime numbers discovered by various invocations for different test numbers. Using this approach, each task will benefit from the work done by previous tasks. Be careful to protect this list with `lock` regions, so access to the list by different threads is serialized.

You can also improve this example by testing for trivial divisors, like 2, 3, and 5.

See Also

How to: Run an Operation in the Background
Event-based Asynchronous Pattern Overview
NOT IN BUILD: Multithreading in Visual Basic
How to: Implement a Component That Supports the Event-based Asynchronous Pattern
Multithreaded Programming with the Event-based Asynchronous Pattern

How to: Implement a Client of the Event-based Asynchronous Pattern

11/21/2017 • 26 min to read • Edit Online

The following code example demonstrates how to use a component that adheres to the Event-based Asynchronous Pattern Overview. The form for this example uses the `PrimeNumberCalculator` component described in How to: Implement a Component That Supports the Event-based Asynchronous Pattern.

When you run a project that uses this example, you will see a "Prime Number Calculator" form with a grid and two buttons: **Start New Task** and **Cancel**. You can click the **Start New Task** button several times in succession, and for each click, an asynchronous operation will begin a computation to determine if a randomly generated test number is prime. The form will periodically display progress and incremental results. Each operation is assigned a unique task ID. The result of the computation is displayed in the **Result** column; if the test number is not prime, it is labeled as **Composite,** and its first divisor is displayed.

Any pending operation can be canceled with the **Cancel** button. Multiple selections can be made.

> **NOTE**
> Most numbers will not be prime. If you have not found a prime number after several completed operations, simply start more tasks, and eventually you will find some prime numbers.

Example

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Globalization;
using System.Threading;
using System.Windows.Forms;

namespace AsyncOperationManagerExample
{
    // This form tests the PrimeNumberCalculator component.
    public class PrimeNumberCalculatorMain : System.Windows.Forms.Form
 {
        //////////////////////////////////////////////////////////
  // Private fields
        //
        #region Private fields

        private PrimeNumberCalculator primeNumberCalculator1;
        private System.Windows.Forms.GroupBox taskGroupBox;
        private System.Windows.Forms.ListView listView1;
        private System.Windows.Forms.ColumnHeader taskIdColHeader;
        private System.Windows.Forms.ColumnHeader progressColHeader;
        private System.Windows.Forms.ColumnHeader currentColHeader;
        private System.Windows.Forms.Panel buttonPanel;
        private System.Windows.Forms.Panel panel2;
        private System.Windows.Forms.Button startAsyncButton;
        private System.Windows.Forms.Button cancelButton;
        private System.Windows.Forms.ColumnHeader testNumberColHeader;
```

```
        private System.Windows.Forms.ColumnHeader resultColHeader;
        private System.Windows.Forms.ColumnHeader firstDivisorColHeader;
        private System.ComponentModel.IContainer components;
        private int progressCounter;
        private int progressInterval = 100;


        #endregion // Private fields

        //////////////////////////////////////////////////////////
        // Construction and destruction
        //
        #region Private fields
public PrimeNumberCalculatorMain ()
{
 //
 // Required for Windows Form Designer support
 //
 InitializeComponent();

          // Hook up event handlers.
          this.primeNumberCalculator1.CalculatePrimeCompleted +=
              new CalculatePrimeCompletedEventHandler(
              primeNumberCalculator1_CalculatePrimeCompleted);

          this.primeNumberCalculator1.ProgressChanged +=
              new ProgressChangedEventHandler(
              primeNumberCalculator1_ProgressChanged);

          this.listView1.SelectedIndexChanged +=
              new EventHandler(listView1_SelectedIndexChanged);
}


protected override void Dispose( bool disposing )
{
 if( disposing )
 {
  if (components != null)
  {
   components.Dispose();
  }
 }
 base.Dispose( disposing );
}

        #endregion // Construction and destruction


        //////////////////////////////////////////////////////////
        //
        #region Implementation

        // This event handler selects a number randomly to test
        // for primality. It then starts the asynchronous
        // calculation by calling the PrimeNumberCalculator
        // component's CalculatePrimeAsync method.
        private void startAsyncButton_Click (
            System.Object sender, System.EventArgs e)
        {
            // Randomly choose test numbers
            // up to 200,000 for primality.
            Random rand = new Random();
            int testNumber = rand.Next(200000);

            // Task IDs are Guids.
            Guid taskId = Guid.NewGuid();
            this.AddListViewItem(taskId, testNumber);
```

```
        // Start the asynchronous task.
        this.primeNumberCalculator1.CalculatePrimeAsync(
            testNumber,
            taskId);
    }

    private void listView1_SelectedIndexChanged(
        object sender,
        EventArgs e)
    {
        this.cancelButton.Enabled = CanCancel();
    }

    // This event handler cancels all pending tasks that are
    // selected in the ListView control.
    private void cancelButton_Click(
        System.Object sender,
        System.EventArgs e)
    {
        Guid taskId = Guid.Empty;

        // Cancel all selected tasks.
        foreach(ListViewItem lvi in this.listView1.SelectedItems)
        {
            // Tasks that have been completed or canceled have
            // their corresponding ListViewItem.Tag property
            // set to null.
            if (lvi.Tag != null)
            {
                taskId = (Guid)lvi.Tag;
                this.primeNumberCalculator1.CancelAsync(taskId);
                lvi.Selected = false;
            }
        }

        cancelButton.Enabled = false;
    }

    // This event handler updates the ListView control when the
    // PrimeNumberCalculator raises the ProgressChanged event.
    //
    // On fast computers, the PrimeNumberCalculator can raise many
    // successive ProgressChanged events, so the user interface
    // may be flooded with messages. To prevent the user interface
    // from hanging, progress is only reported at intervals.
    private void primeNumberCalculator1_ProgressChanged(
        ProgressChangedEventArgs e)
    {
        if (this.progressCounter++ % this.progressInterval == 0)
        {
            Guid taskId = (Guid)e.UserState;

            if (e is CalculatePrimeProgressChangedEventArgs)
            {
                CalculatePrimeProgressChangedEventArgs cppcea =
                    e as CalculatePrimeProgressChangedEventArgs;

                this.UpdateListViewItem(
                    taskId,
                    cppcea.ProgressPercentage,
                    cppcea.LatestPrimeNumber);
            }
            else
            {
                this.UpdateListViewItem(
                    taskId,
                    e.ProgressPercentage);
            }
        }
```

```
            else if (this.progressCounter > this.progressInterval)
            {
                this.progressCounter = 0;
            }
        }

        // This event handler updates the ListView control when the
        // PrimeNumberCalculator raises the CalculatePrimeCompleted
        // event. The ListView item is updated with the appropriate
        // outcome of the calculation: Canceled, Error, or result.
        private void primeNumberCalculator1_CalculatePrimeCompleted(
            object sender,
            CalculatePrimeCompletedEventArgs e)
        {
            Guid taskId = (Guid)e.UserState;

            if (e.Cancelled)
            {
                string result = "Canceled";

                ListViewItem lvi = UpdateListViewItem(taskId, result);

                if (lvi != null)
                {
                    lvi.BackColor = Color.Pink;
                    lvi.Tag = null;
                }
            }
            else if (e.Error != null)
            {
                string result = "Error";

                ListViewItem lvi = UpdateListViewItem(taskId, result);

                if (lvi != null)
                {
                    lvi.BackColor = Color.Red;
                    lvi.ForeColor = Color.White;
                    lvi.Tag = null;
                }
            }
            else
            {
                bool result = e.IsPrime;

                ListViewItem lvi = UpdateListViewItem(
                    taskId,
                    result,
                    e.FirstDivisor);

                if (lvi != null)
                {
                    lvi.BackColor = Color.LightGray;
                    lvi.Tag = null;
                }
            }
        }

        #endregion // Implementation

        //////////////////////////////////////////////////////////
        //
        #region Private Methods

        private ListViewItem AddListViewItem(
            Guid guid,
            int testNumber )
        {
            ListViewItem lvi = new ListViewItem();
```

```
            lvi.Text = testNumber.ToString(
                CultureInfo.CurrentCulture.NumberFormat);

            lvi.SubItems.Add("Not Started");
            lvi.SubItems.Add("1");
            lvi.SubItems.Add(guid.ToString());
            lvi.SubItems.Add("---");
            lvi.SubItems.Add("---");
            lvi.Tag = guid;

            this.listView1.Items.Add( lvi );

            return lvi;
        }

        private ListViewItem UpdateListViewItem(
            Guid guid,
            int percentComplete,
            int current )
        {
            ListViewItem lviRet = null;

            foreach (ListViewItem lvi in this.listView1.Items)
            {
                if (lvi.Tag != null)
                {
                    if ((Guid)lvi.Tag == guid)
                    {
                        lvi.SubItems[1].Text =
                            percentComplete.ToString(
                            CultureInfo.CurrentCulture.NumberFormat);
                        lvi.SubItems[2].Text =
                            current.ToString(
                            CultureInfo.CurrentCulture.NumberFormat);
                        lviRet = lvi;
                        break;
                    }
                }
            }

            return lviRet;
        }

        private ListViewItem UpdateListViewItem(
            Guid guid,
            int percentComplete,
            int current,
            bool result,
            int firstDivisor )
        {
            ListViewItem lviRet = null;

            foreach (ListViewItem lvi in this.listView1.Items)
            {
                if ((Guid)lvi.Tag == guid)
                {
                    lvi.SubItems[1].Text =
                        percentComplete.ToString(
                        CultureInfo.CurrentCulture.NumberFormat);
                    lvi.SubItems[2].Text =
                        current.ToString(
                        CultureInfo.CurrentCulture.NumberFormat);
                    lvi.SubItems[4].Text =
                        result ? "Prime" : "Composite";
                    lvi.SubItems[5].Text =
                        firstDivisor.ToString(
                        CultureInfo.CurrentCulture.NumberFormat);

                    lviRet = lvi;
```

```
            break;
        }
    }

    return lviRet;
}

private ListViewItem UpdateListViewItem(
    Guid guid,
    int percentComplete )
{
    ListViewItem lviRet = null;

    foreach (ListViewItem lvi in this.listView1.Items)
    {
        if (lvi.Tag != null)
        {
            if ((Guid)lvi.Tag == guid)
            {
                lvi.SubItems[1].Text =
                    percentComplete.ToString(
                    CultureInfo.CurrentCulture.NumberFormat);
                lviRet = lvi;
                break;
            }
        }
    }

    return lviRet;
}

private ListViewItem UpdateListViewItem(
    Guid guid,
    bool result,
    int firstDivisor )
{
    ListViewItem lviRet = null;

    foreach (ListViewItem lvi in this.listView1.Items)
    {
        if (lvi.Tag != null)
        {
            if ((Guid)lvi.Tag == guid)
            {
                lvi.SubItems[4].Text =
                    result ? "Prime" : "Composite";
                lvi.SubItems[5].Text =
                    firstDivisor.ToString(
                    CultureInfo.CurrentCulture.NumberFormat);
                lviRet = lvi;
                break;
            }
        }
    }

    return lviRet;
}

private ListViewItem UpdateListViewItem(
    Guid guid,
    string result)
{
    ListViewItem lviRet = null;

    foreach (ListViewItem lvi in this.listView1.Items)
    {
        if (lvi.Tag != null)
        {
```

```
                {
                    if ((Guid)lvi.Tag == guid)
                    {
                        lvi.SubItems[4].Text = result;
                        lviRet = lvi;
                        break;
                    }
                }
            }

            return lviRet;
        }

        private bool CanCancel()
        {
            bool oneIsActive = false;

            foreach(ListViewItem lvi in this.listView1.SelectedItems)
            {
                if (lvi.Tag != null)
                {
                    oneIsActive = true;
                    break;
                }
            }

            return( oneIsActive == true );
        }

        #endregion

  #region Windows Form Designer generated code

  private void InitializeComponent()
  {
            this.components = new System.ComponentModel.Container();
            this.taskGroupBox = new System.Windows.Forms.GroupBox();
            this.buttonPanel = new System.Windows.Forms.Panel();
            this.cancelButton = new System.Windows.Forms.Button();
            this.startAsyncButton = new System.Windows.Forms.Button();
            this.listView1 = new System.Windows.Forms.ListView();
            this.testNumberColHeader = new System.Windows.Forms.ColumnHeader();
            this.progressColHeader = new System.Windows.Forms.ColumnHeader();
            this.currentColHeader = new System.Windows.Forms.ColumnHeader();
            this.taskIdColHeader = new System.Windows.Forms.ColumnHeader();
            this.resultColHeader = new System.Windows.Forms.ColumnHeader();
            this.firstDivisorColHeader = new System.Windows.Forms.ColumnHeader();
            this.panel2 = new System.Windows.Forms.Panel();
            this.primeNumberCalculator1 = new
AsyncOperationManagerExample.PrimeNumberCalculator(this.components);
            this.taskGroupBox.SuspendLayout();
            this.buttonPanel.SuspendLayout();
            this.SuspendLayout();
            //
            // taskGroupBox
            //
            this.taskGroupBox.Controls.Add(this.buttonPanel);
            this.taskGroupBox.Controls.Add(this.listView1);
            this.taskGroupBox.Dock = System.Windows.Forms.DockStyle.Fill;
            this.taskGroupBox.Location = new System.Drawing.Point(0, 0);
            this.taskGroupBox.Name = "taskGroupBox";
            this.taskGroupBox.Size = new System.Drawing.Size(608, 254);
            this.taskGroupBox.TabIndex = 1;
            this.taskGroupBox.TabStop = false;
            this.taskGroupBox.Text = "Tasks";
            //
            // buttonPanel
            //
            this.buttonPanel.Controls.Add(this.cancelButton);
            this.buttonPanel.Controls.Add(this.startAsyncButton);
```

```
this.buttonPanel.Controls.Add(this.startAsyncButton);
this.buttonPanel.Dock = System.Windows.Forms.DockStyle.Bottom;
this.buttonPanel.Location = new System.Drawing.Point(3, 176);
this.buttonPanel.Name = "buttonPanel";
this.buttonPanel.Size = new System.Drawing.Size(602, 75);
this.buttonPanel.TabIndex = 1;
//
// cancelButton
//
this.cancelButton.Enabled = false;
this.cancelButton.Location = new System.Drawing.Point(128, 24);
this.cancelButton.Name = "cancelButton";
this.cancelButton.Size = new System.Drawing.Size(88, 23);
this.cancelButton.TabIndex = 1;
this.cancelButton.Text = "Cancel";
this.cancelButton.Click += new System.EventHandler(this.cancelButton_Click);
//
// startAsyncButton
//
this.startAsyncButton.Location = new System.Drawing.Point(24, 24);
this.startAsyncButton.Name = "startAsyncButton";
this.startAsyncButton.Size = new System.Drawing.Size(88, 23);
this.startAsyncButton.TabIndex = 0;
this.startAsyncButton.Text = "Start New Task";
this.startAsyncButton.Click += new System.EventHandler(this.startAsyncButton_Click);
//
// listView1
//
this.listView1.Columns.AddRange(new System.Windows.Forms.ColumnHeader[] {
    this.testNumberColHeader,
    this.progressColHeader,
    this.currentColHeader,
    this.taskIdColHeader,
    this.resultColHeader,
    this.firstDivisorColHeader});
this.listView1.Dock = System.Windows.Forms.DockStyle.Fill;
this.listView1.FullRowSelect = true;
this.listView1.GridLines = true;
this.listView1.Location = new System.Drawing.Point(3, 16);
this.listView1.Name = "listView1";
this.listView1.Size = new System.Drawing.Size(602, 160);
this.listView1.TabIndex = 0;
this.listView1.View = System.Windows.Forms.View.Details;
//
// testNumberColHeader
//
this.testNumberColHeader.Text = "Test Number";
this.testNumberColHeader.Width = 80;
//
// progressColHeader
//
this.progressColHeader.Text = "Progress";
//
// currentColHeader
//
this.currentColHeader.Text = "Current";
//
// taskIdColHeader
//
this.taskIdColHeader.Text = "Task ID";
this.taskIdColHeader.Width = 200;
//
// resultColHeader
//
this.resultColHeader.Text = "Result";
this.resultColHeader.Width = 80;
//
// firstDivisorColHeader
//
```

```
            this.firstDivisorColHeader.Text = "First Divisor";
            this.firstDivisorColHeader.Width = 80;
            //
            // panel2
            //
            this.panel2.Location = new System.Drawing.Point(200, 128);
            this.panel2.Name = "panel2";
            this.panel2.TabIndex = 2;
            //
            // PrimeNumberCalculatorMain
            //
            this.ClientSize = new System.Drawing.Size(608, 254);
            this.Controls.Add(this.taskGroupBox);
            this.Name = "PrimeNumberCalculatorMain";
            this.Text = "Prime Number Calculator";
            this.taskGroupBox.ResumeLayout(false);
            this.buttonPanel.ResumeLayout(false);
            this.ResumeLayout(false);

        }
#endregion

[STAThread]
static void Main()
{
 Application.Run(new PrimeNumberCalculatorMain());
        }


  }


  ////////////////////////////////////////////////////////////
  #region PrimeNumberCalculator Implementation

  public delegate void ProgressChangedEventHandler(
      ProgressChangedEventArgs e);

  public delegate void CalculatePrimeCompletedEventHandler(
      object sender,
      CalculatePrimeCompletedEventArgs e);

  // This class implements the Event-based Asynchronous Pattern.
  // It asynchronously computes whether a number is prime or
  // composite (not prime).
  public class PrimeNumberCalculator : Component
  {
      private delegate void WorkerEventHandler(
          int numberToCheck,
          AsyncOperation asyncOp);

      private SendOrPostCallback onProgressReportDelegate;
      private SendOrPostCallback onCompletedDelegate;

      private HybridDictionary userStateToLifetime =
          new HybridDictionary();

      private System.ComponentModel.Container components = null;

      ////////////////////////////////////////////////////////////
      #region Public events

      public event ProgressChangedEventHandler ProgressChanged;
      public event CalculatePrimeCompletedEventHandler CalculatePrimeCompleted;

      #endregion

      ////////////////////////////////////////////////////////////
      #region Construction and destruction
```

```
public PrimeNumberCalculator(IContainer container)
{
    container.Add(this);
    InitializeComponent();

    InitializeDelegates();
}

public PrimeNumberCalculator()
{
    InitializeComponent();

    InitializeDelegates();
}

protected virtual void InitializeDelegates()
{
    onProgressReportDelegate =
        new SendOrPostCallback(ReportProgress);
    onCompletedDelegate =
        new SendOrPostCallback(CalculateCompleted);
}

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose(disposing);
}

#endregion // Construction and destruction

//////////////////////////////////////////////////////////////
///
#region Implementation

// This method starts an asynchronous calculation.
// First, it checks the supplied task ID for uniqueness.
// If taskId is unique, it creates a new WorkerEventHandler
// and calls its BeginInvoke method to start the calculation.
public virtual void CalculatePrimeAsync(
    int numberToTest,
    object taskId)
{
    // Create an AsyncOperation for taskId.
    AsyncOperation asyncOp =
        AsyncOperationManager.CreateOperation(taskId);

    // Multiple threads will access the task dictionary,
    // so it must be locked to serialize access.
    lock (userStateToLifetime.SyncRoot)
    {
        if (userStateToLifetime.Contains(taskId))
        {
            throw new ArgumentException(
                "Task ID parameter must be unique",
                "taskId");
        }

        userStateToLifetime[taskId] = asyncOp;
    }

    // Start the asynchronous operation.
```

```
            WorkerEventHandler workerDelegate = new WorkerEventHandler(CalculateWorker);
            workerDelegate.BeginInvoke(
                numberToTest,
                asyncOp,
                null,
                null);
        }

        // Utility method for determining if a
        // task has been canceled.
        private bool TaskCanceled(object taskId)
        {
            return( userStateToLifetime[taskId] == null );
        }

        // This method cancels a pending asynchronous operation.
        public void CancelAsync(object taskId)
        {
            AsyncOperation asyncOp = userStateToLifetime[taskId] as AsyncOperation;
            if (asyncOp != null)
            {
                lock (userStateToLifetime.SyncRoot)
                {
                    userStateToLifetime.Remove(taskId);
                }
            }
        }

        // This method performs the actual prime number computation.
        // It is executed on the worker thread.
        private void CalculateWorker(
            int numberToTest,
            AsyncOperation asyncOp)
        {
            bool isPrime = false;
            int firstDivisor = 1;
            Exception e = null;

            // Check that the task is still active.
            // The operation may have been canceled before
            // the thread was scheduled.
            if (!TaskCanceled(asyncOp.UserSuppliedState))
            {
                try
                {
                    // Find all the prime numbers up to
                    // the square root of numberToTest.
                    ArrayList primes = BuildPrimeNumberList(
                        numberToTest,
                        asyncOp);

                    // Now we have a list of primes less than
                    // numberToTest.
                    isPrime = IsPrime(
                        primes,
                        numberToTest,
                        out firstDivisor);
                }
                catch (Exception ex)
                {
                    e = ex;
                }
            }

            //CalculatePrimeState calcState = new CalculatePrimeState(
            //        numberToTest,
            //        firstDivisor,
            //        isPrime,
            //        e,
```

```
            //        TaskCanceled(asyncOp.UserSuppliedState),
            //        asyncOp);

            //this.CompletionMethod(calcState);

            this.CompletionMethod(
                numberToTest,
                firstDivisor,
                isPrime,
                e,
                TaskCanceled(asyncOp.UserSuppliedState),
                asyncOp);

            //completionMethodDelegate(calcState);
        }

        // This method computes the list of prime numbers used by the
        // IsPrime method.
        private ArrayList BuildPrimeNumberList(
            int numberToTest,
            AsyncOperation asyncOp)
        {
            ProgressChangedEventArgs e = null;
            ArrayList primes = new ArrayList();
            int firstDivisor;
            int n = 5;

            // Add the first prime numbers.
            primes.Add(2);
            primes.Add(3);

            // Do the work.
            while (n < numberToTest &&
                   !TaskCanceled( asyncOp.UserSuppliedState ) )
            {
                if (IsPrime(primes, n, out firstDivisor))
                {
                    // Report to the client that a prime was found.
                    e = new CalculatePrimeProgressChangedEventArgs(
                        n,
                        (int)((float)n / (float)numberToTest * 100),
                        asyncOp.UserSuppliedState);

                    asyncOp.Post(this.onProgressReportDelegate, e);

                    primes.Add(n);

                    // Yield the rest of this time slice.
                    Thread.Sleep(0);
                }

                // Skip even numbers.
                n += 2;
            }

            return primes;
        }

        // This method tests n for primality against the list of
        // prime numbers contained in the primes parameter.
        private bool IsPrime(
            ArrayList primes,
            int n,
            out int firstDivisor)
        {
            bool foundDivisor = false;
            bool exceedsSquareRoot = false;

            int i = 0;
```

```
        int i = 0;
        int divisor = 0;
        firstDivisor = 1;

        // Stop the search if:
        // there are no more primes in the list,
        // there is a divisor of n in the list, or
        // there is a prime that is larger than
        // the square root of n.
        while (
            (i < primes.Count) &&
            !foundDivisor &&
            !exceedsSquareRoot)
        {
            // The divisor variable will be the smallest
            // prime number not yet tried.
            divisor = (int)primes[i++];

            // Determine whether the divisor is greater
            // than the square root of n.
            if (divisor * divisor > n)
            {
                exceedsSquareRoot = true;
            }
            // Determine whether the divisor is a factor of n.
            else if (n % divisor == 0)
            {
                firstDivisor = divisor;
                foundDivisor = true;
            }
        }

        return !foundDivisor;
    }

    // This method is invoked via the AsyncOperation object,
    // so it is guaranteed to be executed on the correct thread.
    private void CalculateCompleted(object operationState)
    {
        CalculatePrimeCompletedEventArgs e =
            operationState as CalculatePrimeCompletedEventArgs;

        OnCalculatePrimeCompleted(e);
    }

    // This method is invoked via the AsyncOperation object,
    // so it is guaranteed to be executed on the correct thread.
    private void ReportProgress(object state)
    {
        ProgressChangedEventArgs e =
            state as ProgressChangedEventArgs;

        OnProgressChanged(e);
    }

    protected void OnCalculatePrimeCompleted(
        CalculatePrimeCompletedEventArgs e)
    {
        if (CalculatePrimeCompleted != null)
        {
            CalculatePrimeCompleted(this, e);
        }
    }

    protected void OnProgressChanged(ProgressChangedEventArgs e)
    {
        if (ProgressChanged != null)
        {
            ProgressChanged(e);
        }
```

```
        }
    }

    // This is the method that the underlying, free-threaded
    // asynchronous behavior will invoke.  This will happen on
    // an arbitrary thread.
    private void CompletionMethod(
        int numberToTest,
        int firstDivisor,
        bool isPrime,
        Exception exception,
        bool canceled,
        AsyncOperation asyncOp )

    {
        // If the task was not previously canceled,
        // remove the task from the lifetime collection.
        if (!canceled)
        {
            lock (userStateToLifetime.SyncRoot)
            {
                userStateToLifetime.Remove(asyncOp.UserSuppliedState);
            }
        }

        // Package the results of the operation in a
        // CalculatePrimeCompletedEventArgs.
        CalculatePrimeCompletedEventArgs e =
            new CalculatePrimeCompletedEventArgs(
            numberToTest,
            firstDivisor,
            isPrime,
            exception,
            canceled,
            asyncOp.UserSuppliedState);

        // End the task. The asyncOp object is responsible
        // for marshaling the call.
        asyncOp.PostOperationCompleted(onCompletedDelegate, e);

        // Note that after the call to OperationCompleted,
        // asyncOp is no longer usable, and any attempt to use it
        // will cause an exception to be thrown.
    }


    #endregion

    //////////////////////////////////////////////////////////////
    #region Component Designer generated code

    private void InitializeComponent()
    {
        components = new System.ComponentModel.Container();
    }

    #endregion

}

public class CalculatePrimeProgressChangedEventArgs :
        ProgressChangedEventArgs
{
    private int latestPrimeNumberValue = 1;

    public CalculatePrimeProgressChangedEventArgs(
        int latestPrime,
        int progressPercentage,
        object userToken) : base( progressPercentage, userToken )
    {
```

```
        {
            this.latestPrimeNumberValue = latestPrime;
        }

        public int LatestPrimeNumber
        {
            get
            {
                return latestPrimeNumberValue;
            }
        }
    }

    public class CalculatePrimeCompletedEventArgs :
        AsyncCompletedEventArgs
    {
        private int numberToTestValue = 0;
        private int firstDivisorValue = 1;
        private bool isPrimeValue;

        public CalculatePrimeCompletedEventArgs(
            int numberToTest,
            int firstDivisor,
            bool isPrime,
            Exception e,
            bool canceled,
            object state) : base(e, canceled, state)
        {
            this.numberToTestValue = numberToTest;
            this.firstDivisorValue = firstDivisor;
            this.isPrimeValue = isPrime;
        }

        public int NumberToTest
        {
            get
            {
                // Raise an exception if the operation failed or
                // was canceled.
                RaiseExceptionIfNecessary();

                // If the operation was successful, return the
                // property value.
                return numberToTestValue;
            }
        }

        public int FirstDivisor
        {
            get
            {
                // Raise an exception if the operation failed or
                // was canceled.
                RaiseExceptionIfNecessary();

                // If the operation was successful, return the
                // property value.
                return firstDivisorValue;
            }
        }

        public bool IsPrime
        {
            get
            {
                // Raise an exception if the operation failed or
                // was canceled.
                RaiseExceptionIfNecessary();
```

```
                // If the operation was successful, return the
                // property value.
                return isPrimeValue;
            }
        }
    }


    #endregion


}
```

```
Imports System
Imports System.Collections
Imports System.Collections.Specialized
Imports System.ComponentModel
Imports System.Drawing
Imports System.Globalization
Imports System.Threading
Imports System.Windows.Forms

' This form tests the PrimeNumberCalculator component.
Public Class PrimeNumberCalculatorMain
    Inherits System.Windows.Forms.Form

    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
    ' Private fields
    '
    #Region "Private fields"

    Private WithEvents primeNumberCalculator1 As PrimeNumberCalculator
    Private taskGroupBox As System.Windows.Forms.GroupBox
    Private WithEvents listView1 As System.Windows.Forms.ListView
    Private taskIdColHeader As System.Windows.Forms.ColumnHeader
    Private progressColHeader As System.Windows.Forms.ColumnHeader
    Private currentColHeader As System.Windows.Forms.ColumnHeader
    Private buttonPanel As System.Windows.Forms.Panel
    Private panel2 As System.Windows.Forms.Panel
    Private WithEvents startAsyncButton As System.Windows.Forms.Button
    Private WithEvents cancelAsyncButton As System.Windows.Forms.Button
    Private testNumberColHeader As System.Windows.Forms.ColumnHeader
    Private resultColHeader As System.Windows.Forms.ColumnHeader
    Private firstDivisorColHeader As System.Windows.Forms.ColumnHeader
    Private components As System.ComponentModel.IContainer
    Private progressCounter As Integer
    Private progressInterval As Integer = 100


#End Region


    Public Sub New()

        InitializeComponent()

    End Sub

    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If (components IsNot Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)

    End Sub
```

```
    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
#Region "Implementation"

    ' This event handler selects a number randomly to test
    ' for primality. It then starts the asynchronous
    ' calculation by calling the PrimeNumberCalculator
    ' component's CalculatePrimeAsync method.
    Private Sub startAsyncButton_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles startAsyncButton.Click

        ' Randomly choose test numbers
        ' up to 200,000 for primality.
        Dim rand As New Random
        Dim testNumber As Integer = rand.Next(200000)

        ' Task IDs are Guids.
        Dim taskId As Guid = Guid.NewGuid()
        Me.AddListViewItem(taskId, testNumber)

        ' Start the asynchronous task.
        Me.primeNumberCalculator1.CalculatePrimeAsync( _
        testNumber, _
        taskId)

    End Sub


    Private Sub listView1_SelectedIndexChanged( _
        ByVal sender As Object, ByVal e As EventArgs) _
        Handles listView1.SelectedIndexChanged

        Me.cancelAsyncButton.Enabled = CanCancel()

    End Sub


    ' This event handler cancels all pending tasks that are
    ' selected in the ListView control.
    Private Sub cancelAsyncButton_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles cancelAsyncButton.Click

        Dim taskId As Guid = Guid.Empty

        ' Cancel all selected tasks.
        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.SelectedItems
            ' Tasks that have been completed or canceled have
            ' their corresponding ListViewItem.Tag property
            ' set to Nothing.
            If (lvi.Tag IsNot Nothing) Then
                taskId = CType(lvi.Tag, Guid)
                Me.primeNumberCalculator1.CancelAsync(taskId)
                lvi.Selected = False
            End If
        Next lvi

        cancelAsyncButton.Enabled = False

    End Sub

    ' This event handler updates the ListView control when the
    ' PrimeNumberCalculator raises the ProgressChanged event.
    '
```

```
    ' On fast computers, the PrimeNumberCalculator can raise many
    ' successive ProgressChanged events, so the user interface
    ' may be flooded with messages. To prevent the user interface
    ' from hanging, progress is only reported at intervals.
    Private Sub primeNumberCalculator1_ProgressChanged( _
        ByVal e As ProgressChangedEventArgs) _
        Handles primeNumberCalculator1.ProgressChanged

        Me.progressCounter += 1

        If Me.progressCounter Mod Me.progressInterval = 0 Then

            Dim taskId As Guid = CType(e.UserState, Guid)

            If TypeOf e Is CalculatePrimeProgressChangedEventArgs Then
                Dim cppcea As CalculatePrimeProgressChangedEventArgs = e
                Me.UpdateListViewItem( _
                    taskId, _
                    cppcea.ProgressPercentage, _
                    cppcea.LatestPrimeNumber)
            Else
                Me.UpdateListViewItem( _
                    taskId, e.ProgressPercentage)
            End If
        ElseIf Me.progressCounter > Me.progressInterval Then
            Me.progressCounter = 0
        End If

    End Sub

    ' This event handler updates the ListView control when the
    ' PrimeNumberCalculator raises the CalculatePrimeCompleted
    ' event. The ListView item is updated with the appropriate
    ' outcome of the calculation: Canceled, Error, or result.
    Private Sub primeNumberCalculator1_CalculatePrimeCompleted( _
        ByVal sender As Object, _
        ByVal e As CalculatePrimeCompletedEventArgs) _
        Handles primeNumberCalculator1.CalculatePrimeCompleted

        Dim taskId As Guid = CType(e.UserState, Guid)

        If e.Cancelled Then
            Dim result As String = "Canceled"

            Dim lvi As ListViewItem = UpdateListViewItem( _
                taskId, _
                result)

            If (lvi IsNot Nothing) Then
                lvi.BackColor = Color.Pink
                lvi.Tag = Nothing
            End If

        ElseIf e.Error IsNot Nothing Then

            Dim result As String = "Error"

            Dim lvi As ListViewItem = UpdateListViewItem( _
                taskId, result)

            If (lvi IsNot Nothing) Then
                lvi.BackColor = Color.Red
                lvi.ForeColor = Color.White
                lvi.Tag = Nothing
            End If
        Else
            Dim result As Boolean = e.IsPrime

            Dim lvi As ListViewItem = UpdateListViewItem( _
```

```
                taskId, _
                result, _
                e.FirstDivisor)

            If (lvi IsNot Nothing) Then
                lvi.BackColor = Color.LightGray
                lvi.Tag = Nothing
            End If
        End If

    End Sub

#End Region

    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
#Region "Private Methods"

    Private Function AddListViewItem( _
        ByVal guid As Guid, _
        ByVal testNumber As Integer) As ListViewItem

        Dim lvi As New ListViewItem
        lvi.Text = testNumber.ToString( _
        CultureInfo.CurrentCulture.NumberFormat)

        lvi.SubItems.Add("Not Started")
        lvi.SubItems.Add("1")
        lvi.SubItems.Add(guid.ToString())
        lvi.SubItems.Add("---")
        lvi.SubItems.Add("---")
        lvi.Tag = guid

        Me.listView1.Items.Add(lvi)

        Return lvi

    End Function

    Private Overloads Function UpdateListViewItem( _
        ByVal guid As Guid, _
        ByVal percentComplete As Integer, _
        ByVal current As Integer) As ListViewItem

        Dim lviRet As ListViewItem = Nothing

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.Items
            If (lvi.Tag IsNot Nothing) Then
                If guid.CompareTo(CType(lvi.Tag, Guid)) = 0 Then
                    lvi.SubItems(1).Text = percentComplete.ToString( _
                    CultureInfo.CurrentCulture.NumberFormat)
                    lvi.SubItems(2).Text = current.ToString( _
                    CultureInfo.CurrentCulture.NumberFormat)
                    lviRet = lvi
                    Exit For
                End If
            End If
        Next lvi

        Return lviRet

    End Function


    Private Overloads Function UpdateListViewItem( _
        ByVal guid As Guid, _
        ByVal percentComplete As Integer, _
        ByVal current As Integer, _
        ByVal result As Boolean, _
```

```
        ByVal firstDivisor As Integer) As ListViewItem

        Dim lviRet As ListViewItem = Nothing

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.Items
            If guid.CompareTo(CType(lvi.Tag, Guid)) = 0 Then
                lvi.SubItems(1).Text = percentComplete.ToString( _
                CultureInfo.CurrentCulture.NumberFormat)
                lvi.SubItems(2).Text = current.ToString( _
                CultureInfo.CurrentCulture.NumberFormat)
                lvi.SubItems(4).Text = _
                    IIf(result, "Prime", "Composite")
                lvi.SubItems(5).Text = firstDivisor.ToString( _
                CultureInfo.CurrentCulture.NumberFormat)

                lviRet = lvi

                Exit For
            End If
        Next lvi

        Return lviRet

    End Function


    Private Overloads Function UpdateListViewItem( _
        ByVal guid As Guid, _
        ByVal percentComplete As Integer) As ListViewItem

        Dim lviRet As ListViewItem = Nothing

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.Items
            If (lvi.Tag IsNot Nothing) Then
                If guid.CompareTo(CType(lvi.Tag, Guid)) = 0 Then
                    lvi.SubItems(1).Text = percentComplete.ToString( _
                    CultureInfo.CurrentCulture.NumberFormat)
                    lviRet = lvi
                    Exit For
                End If
            End If
        Next lvi

        Return lviRet

    End Function


    Private Overloads Function UpdateListViewItem( _
        ByVal guid As Guid, _
        ByVal result As Boolean, _
        ByVal firstDivisor As Integer) As ListViewItem

        Dim lviRet As ListViewItem = Nothing

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.Items
            If (lvi.Tag IsNot Nothing) Then
                If guid.CompareTo(CType(lvi.Tag, Guid)) = 0 Then
                    lvi.SubItems(4).Text = _
                        IIf(result, "Prime", "Composite")
                    lvi.SubItems(5).Text = firstDivisor.ToString( _
                    CultureInfo.CurrentCulture.NumberFormat)
                    lviRet = lvi
                    Exit For
                End If
            End If
```

```
            End If
        Next lvi

        Return lviRet

    End Function


    Private Overloads Function UpdateListViewItem( _
        ByVal guid As Guid, _
        ByVal result As String) As ListViewItem

        Dim lviRet As ListViewItem = Nothing

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.Items
            If (lvi.Tag IsNot Nothing) Then
                If guid.CompareTo(CType(lvi.Tag, Guid)) = 0 Then
                    lvi.SubItems(4).Text = result
                    lviRet = lvi
                    Exit For
                End If
            End If
        Next lvi

        Return lviRet

    End Function


    Private Function CanCancel() As Boolean
        Dim oneIsActive As Boolean = False

        Dim lvi As ListViewItem
        For Each lvi In Me.listView1.SelectedItems
            If (lvi.Tag IsNot Nothing) Then
                oneIsActive = True
                Exit For
            End If
        Next lvi

        Return oneIsActive = True

    End Function

#End Region


    Private Sub InitializeComponent()
        Me.components = New System.ComponentModel.Container
        Me.taskGroupBox = New System.Windows.Forms.GroupBox
        Me.buttonPanel = New System.Windows.Forms.Panel
        Me.cancelAsyncButton = New System.Windows.Forms.Button
        Me.startAsyncButton = New System.Windows.Forms.Button
        Me.listView1 = New System.Windows.Forms.ListView
        Me.testNumberColHeader = New System.Windows.Forms.ColumnHeader
        Me.progressColHeader = New System.Windows.Forms.ColumnHeader
        Me.currentColHeader = New System.Windows.Forms.ColumnHeader
        Me.taskIdColHeader = New System.Windows.Forms.ColumnHeader
        Me.resultColHeader = New System.Windows.Forms.ColumnHeader
        Me.firstDivisorColHeader = New System.Windows.Forms.ColumnHeader
        Me.panel2 = New System.Windows.Forms.Panel
        Me.primeNumberCalculator1 = New PrimeNumberCalculator(Me.components)
        Me.taskGroupBox.SuspendLayout()
        Me.buttonPanel.SuspendLayout()
        Me.SuspendLayout()
        '
        ' taskGroupBox
        '
        Me.taskGroupBox.Controls.Add(Me.buttonPanel)
```

```
        Me.taskGroupBox.Controls.Add(Me.buttonPanel)
        Me.taskGroupBox.Controls.Add(Me.listView1)
        Me.taskGroupBox.Dock = System.Windows.Forms.DockStyle.Fill
        Me.taskGroupBox.Location = New System.Drawing.Point(0, 0)
        Me.taskGroupBox.Name = "taskGroupBox"
        Me.taskGroupBox.Size = New System.Drawing.Size(608, 254)
        Me.taskGroupBox.TabIndex = 1
        Me.taskGroupBox.TabStop = False
        Me.taskGroupBox.Text = "Tasks"
        '
        ' buttonPanel
        '
        Me.buttonPanel.Controls.Add(Me.cancelAsyncButton)
        Me.buttonPanel.Controls.Add(Me.startAsyncButton)
        Me.buttonPanel.Dock = System.Windows.Forms.DockStyle.Bottom
        Me.buttonPanel.Location = New System.Drawing.Point(3, 176)
        Me.buttonPanel.Name = "buttonPanel"
        Me.buttonPanel.Size = New System.Drawing.Size(602, 75)
        Me.buttonPanel.TabIndex = 1
        '
        ' cancelAsyncButton
        '
        Me.cancelAsyncButton.Enabled = False
        Me.cancelAsyncButton.Location = New System.Drawing.Point(128, 24)
        Me.cancelAsyncButton.Name = "cancelAsyncButton"
        Me.cancelAsyncButton.Size = New System.Drawing.Size(88, 23)
        Me.cancelAsyncButton.TabIndex = 1
        Me.cancelAsyncButton.Text = "Cancel"
        '
        ' startAsyncButton
        '
        Me.startAsyncButton.Location = New System.Drawing.Point(24, 24)
        Me.startAsyncButton.Name = "startAsyncButton"
        Me.startAsyncButton.Size = New System.Drawing.Size(88, 23)
        Me.startAsyncButton.TabIndex = 0
        Me.startAsyncButton.Text = "Start New Task"
        '
        ' listView1
        '
        Me.listView1.Columns.AddRange(New System.Windows.Forms.ColumnHeader() {Me.testNumberColHeader,
Me.progressColHeader, Me.currentColHeader, Me.taskIdColHeader, Me.resultColHeader, Me.firstDivisorColHeader})
        Me.listView1.Dock = System.Windows.Forms.DockStyle.Fill
        Me.listView1.FullRowSelect = True
        Me.listView1.GridLines = True
        Me.listView1.Location = New System.Drawing.Point(3, 16)
        Me.listView1.Name = "listView1"
        Me.listView1.Size = New System.Drawing.Size(602, 160)
        Me.listView1.TabIndex = 0
        Me.listView1.View = System.Windows.Forms.View.Details
        '
        ' testNumberColHeader
        '
        Me.testNumberColHeader.Text = "Test Number"
        Me.testNumberColHeader.Width = 80
        '
        ' progressColHeader
        '
        Me.progressColHeader.Text = "Progress"
        '
        ' currentColHeader
        '
        Me.currentColHeader.Text = "Current"
        '
        ' taskIdColHeader
        '
        Me.taskIdColHeader.Text = "Task ID"
        Me.taskIdColHeader.Width = 200
        '
        ' resultColHeader
        '
```

```
        '
        Me.resultColHeader.Text = "Result"
        Me.resultColHeader.Width = 80
        '
        ' firstDivisorColHeader
        '
        Me.firstDivisorColHeader.Text = "First Divisor"
        Me.firstDivisorColHeader.Width = 80
        '
        ' panel2
        '
        Me.panel2.Location = New System.Drawing.Point(200, 128)
        Me.panel2.Name = "panel2"
        Me.panel2.TabIndex = 2
        '
        ' PrimeNumberCalculatorMain
        '
        Me.ClientSize = New System.Drawing.Size(608, 254)
        Me.Controls.Add(taskGroupBox)
        Me.Name = "PrimeNumberCalculatorMain"
        Me.Text = "Prime Number Calculator"
        Me.taskGroupBox.ResumeLayout(False)
        Me.buttonPanel.ResumeLayout(False)
        Me.ResumeLayout(False)

    End Sub


    <STAThread()> _
    Shared Sub Main()
        Application.Run(New PrimeNumberCalculatorMain())

    End Sub
End Class


Public Delegate Sub ProgressChangedEventHandler( _
    ByVal e As ProgressChangedEventArgs)

Public Delegate Sub CalculatePrimeCompletedEventHandler( _
    ByVal sender As Object, _
    ByVal e As CalculatePrimeCompletedEventArgs)

' This class implements the Event-based Asynchronous Pattern.
' It asynchronously computes whether a number is prime or
' composite (not prime).
Public Class PrimeNumberCalculator
    Inherits System.ComponentModel.Component

    Private Delegate Sub WorkerEventHandler( _
    ByVal numberToCheck As Integer, _
    ByVal asyncOp As AsyncOperation)

    Private onProgressReportDelegate As SendOrPostCallback
    Private onCompletedDelegate As SendOrPostCallback

    Private userStateToLifetime As New HybridDictionary()

    Private components As System.ComponentModel.Container = Nothing

    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
#Region "Public events"

    Public Event ProgressChanged _
        As ProgressChangedEventHandler
    Public Event CalculatePrimeCompleted _
        As CalculatePrimeCompletedEventHandler

#End Region
```

```
    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
#Region "Construction and destruction"


    Public Sub New(ByVal container As System.ComponentModel.IContainer)

        container.Add(Me)
        InitializeComponent()

        InitializeDelegates()

    End Sub


    Public Sub New()

        InitializeComponent()

        InitializeDelegates()

    End Sub


    Protected Overridable Sub InitializeDelegates()
        onProgressReportDelegate = _
            New SendOrPostCallback(AddressOf ReportProgress)
        onCompletedDelegate = _
            New SendOrPostCallback(AddressOf CalculateCompleted)
    End Sub


    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If (components IsNot Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)

    End Sub

#End Region

    '''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
#Region "Implementation"


    ' This method starts an asynchronous calculation.
    ' First, it checks the supplied task ID for uniqueness.
    ' If taskId is unique, it creates a new WorkerEventHandler
    ' and calls its BeginInvoke method to start the calculation.
    Public Overridable Sub CalculatePrimeAsync( _
        ByVal numberToTest As Integer, _
        ByVal taskId As Object)

        ' Create an AsyncOperation for taskId.
        Dim asyncOp As AsyncOperation = _
            AsyncOperationManager.CreateOperation(taskId)

        ' Multiple threads will access the task dictionary,
        ' so it must be locked to serialize access.
        SyncLock userStateToLifetime.SyncRoot
            If userStateToLifetime.Contains(taskId) Then
                Throw New ArgumentException( _
                    "Task ID parameter must be unique", _
                    "taskId")
            End If
```

```
            userStateToLifetime(taskId) = asyncOp
        End SyncLock

        ' Start the asynchronous operation.
        Dim workerDelegate As New WorkerEventHandler( _
            AddressOf CalculateWorker)

        workerDelegate.BeginInvoke( _
            numberToTest, _
            asyncOp, _
            Nothing, _
            Nothing)

    End Sub

    ' Utility method for determining if a
    ' task has been canceled.
    Private Function TaskCanceled(ByVal taskId As Object) As Boolean
        Return (userStateToLifetime(taskId) Is Nothing)
    End Function

    ' This method cancels a pending asynchronous operation.
    Public Sub CancelAsync(ByVal taskId As Object)

        Dim obj As Object = userStateToLifetime(taskId)
        If (obj IsNot Nothing) Then

            SyncLock userStateToLifetime.SyncRoot

                userStateToLifetime.Remove(taskId)

            End SyncLock

        End If

    End Sub

    ' This method performs the actual prime number computation.
    ' It is executed on the worker thread.
    Private Sub CalculateWorker( _
        ByVal numberToTest As Integer, _
        ByVal asyncOp As AsyncOperation)

        Dim prime As Boolean = False
        Dim firstDivisor As Integer = 1
        Dim exc As Exception = Nothing

        ' Check that the task is still active.
        ' The operation may have been canceled before
        ' the thread was scheduled.
        If Not Me.TaskCanceled(asyncOp.UserSuppliedState) Then

            Try
                ' Find all the prime numbers up to the
                ' square root of numberToTest.
                Dim primes As ArrayList = BuildPrimeNumberList( _
                    numberToTest, asyncOp)

                ' Now we have a list of primes less than
                'numberToTest.
                prime = IsPrime( _
                    primes, _
                    numberToTest, _
                    firstDivisor)

            Catch ex As Exception
                exc = ex
            End Try
```

```
        End If

        Me.CompletionMethod( _
            numberToTest, _
            firstDivisor, _
            prime, _
            exc, _
            TaskCanceled(asyncOp.UserSuppliedState), _
            asyncOp)

    End Sub

    ' This method computes the list of prime numbers used by the
    ' IsPrime method.
    Private Function BuildPrimeNumberList( _
        ByVal numberToTest As Integer, _
        ByVal asyncOp As AsyncOperation) As ArrayList

        Dim e As ProgressChangedEventArgs = Nothing
        Dim primes As New ArrayList
        Dim firstDivisor As Integer
        Dim n As Integer = 5

        ' Add the first prime numbers.
        primes.Add(2)
        primes.Add(3)

        ' Do the work.
        While n < numberToTest And _
            Not Me.TaskCanceled(asyncOp.UserSuppliedState)

            If IsPrime(primes, n, firstDivisor) Then
                ' Report to the client that you found a prime.
                e = New CalculatePrimeProgressChangedEventArgs( _
                    n, _
                    CSng(n) / CSng(numberToTest) * 100, _
                    asyncOp.UserSuppliedState)

                asyncOp.Post(Me.onProgressReportDelegate, e)

                primes.Add(n)

                ' Yield the rest of this time slice.
                Thread.Sleep(0)
            End If

            ' Skip even numbers.
            n += 2

        End While

        Return primes

    End Function


    ' This method tests n for primality against the list of
    ' prime numbers contained in the primes parameter.
    Private Function IsPrime( _
        ByVal primes As ArrayList, _
        ByVal n As Integer, _
        ByRef firstDivisor As Integer) As Boolean

        Dim foundDivisor As Boolean = False
        Dim exceedsSquareRoot As Boolean = False

        Dim i As Integer = 0
        Dim divisor As Integer = 0
        firstDivisor = 1
```

```
    ' Stop the search if:
    ' there are no more primes in the list,
    ' there is a divisor of n in the list, or
    ' there is a prime that is larger than
    ' the square root of n.
    While i < primes.Count AndAlso _
        Not foundDivisor AndAlso _
        Not exceedsSquareRoot

        ' The divisor variable will be the smallest prime number
        ' not yet tried.
        divisor = primes(i)
        i = i + 1

        ' Determine whether the divisor is greater than the
        ' square root of n.
        If divisor * divisor > n Then
            exceedsSquareRoot = True
            ' Determine whether the divisor is a factor of n.
        ElseIf n Mod divisor = 0 Then
            firstDivisor = divisor
            foundDivisor = True
        End If
    End While

    Return Not foundDivisor

End Function


' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub CalculateCompleted(ByVal operationState As Object)
    Dim e As CalculatePrimeCompletedEventArgs = operationState

    OnCalculatePrimeCompleted(e)

End Sub


' This method is invoked via the AsyncOperation object,
' so it is guaranteed to be executed on the correct thread.
Private Sub ReportProgress(ByVal state As Object)
    Dim e As ProgressChangedEventArgs = state

    OnProgressChanged(e)

End Sub

Protected Sub OnCalculatePrimeCompleted( _
    ByVal e As CalculatePrimeCompletedEventArgs)

    RaiseEvent CalculatePrimeCompleted(Me, e)

End Sub


Protected Sub OnProgressChanged( _
    ByVal e As ProgressChangedEventArgs)

    RaiseEvent ProgressChanged(e)

End Sub


' This is the method that the underlying, free-threaded
' asynchronous behavior will invoke.  This will happen on
' an arbitrary thread.
```

```
        an arbitrary thread.
    Private Sub CompletionMethod( _
        ByVal numberToTest As Integer, _
        ByVal firstDivisor As Integer, _
        ByVal prime As Boolean, _
        ByVal exc As Exception, _
        ByVal canceled As Boolean, _
        ByVal asyncOp As AsyncOperation)

        ' If the task was not previously canceled,
        ' remove the task from the lifetime collection.
        If Not canceled Then
            SyncLock userStateToLifetime.SyncRoot
                userStateToLifetime.Remove(asyncOp.UserSuppliedState)
            End SyncLock
        End If

        ' Package the results of the operation in a
        ' CalculatePrimeCompletedEventArgs.
        Dim e As New CalculatePrimeCompletedEventArgs( _
            numberToTest, _
            firstDivisor, _
            prime, _
            exc, _
            canceled, _
            asyncOp.UserSuppliedState)

        ' End the task. The asyncOp object is responsible
        ' for marshaling the call.
        asyncOp.PostOperationCompleted(onCompletedDelegate, e)

        ' Note that after the call to PostOperationCompleted, asyncOp
        ' is no longer usable, and any attempt to use it will cause.
        ' an exception to be thrown.

    End Sub

#End Region


    Private Sub InitializeComponent()

    End Sub


End Class

Public Class CalculatePrimeProgressChangedEventArgs
    Inherits ProgressChangedEventArgs
    Private latestPrimeNumberValue As Integer = 1


    Public Sub New( _
        ByVal latestPrime As Integer, _
        ByVal progressPercentage As Integer, _
        ByVal UserState As Object)

        MyBase.New(progressPercentage, UserState)
        Me.latestPrimeNumberValue = latestPrime

    End Sub

    Public ReadOnly Property LatestPrimeNumber() As Integer
        Get
            Return latestPrimeNumberValue
        End Get
    End Property
End Class

Public Class CalculatePrimeCompletedEventArgs
```

```
Public Class CalculatePrimeCompletedEventArgs
    Inherits AsyncCompletedEventArgs
    Private numberToTestValue As Integer = 0
    Private firstDivisorValue As Integer = 1
    Private isPrimeValue As Boolean


    Public Sub New( _
    ByVal numberToTest As Integer, _
    ByVal firstDivisor As Integer, _
    ByVal isPrime As Boolean, _
    ByVal e As Exception, _
    ByVal canceled As Boolean, _
    ByVal state As Object)

        MyBase.New(e, canceled, state)
        Me.numberToTestValue = numberToTest
        Me.firstDivisorValue = firstDivisor
        Me.isPrimeValue = isPrime

    End Sub


    Public ReadOnly Property NumberToTest() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return numberToTestValue
        End Get
    End Property


    Public ReadOnly Property FirstDivisor() As Integer
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return firstDivisorValue
        End Get
    End Property


    Public ReadOnly Property IsPrime() As Boolean
        Get
            ' Raise an exception if the operation failed
            ' or was canceled.
            RaiseExceptionIfNecessary()

            ' If the operation was successful, return
            ' the property value.
            Return isPrimeValue
        End Get
    End Property
End Class
```

See Also

AsyncOperation
AsyncOperationManager

WindowsFormsSynchronizationContext

How to: Use Components That Support the Event-based Asynchronous Pattern

11/21/2017 • 1 min to read • Edit Online

Many components provide you with the option of performing their work asynchronously. The SoundPlayer and PictureBox components, for example, enable you to load sounds and images "in the background" while your main thread continues running without interruption.

Using asynchronous methods on a class that supports the Event-based Asynchronous Pattern Overview can be as simple as attaching an event handler to the component's *MethodName* `Completed` event, just as you would for any other event. When you call the *MethodName* `Async` method, your application will continue running without interruption until the *MethodName* `Completed` event is raised. In your event handler, you can examine the AsyncCompletedEventArgs parameter to determine if the asynchronous operation successfully completed or if it was canceled.

For more information about using event handlers, see Event Handlers Overview.

The following procedure shows how to use the asynchronous image-loading capability of a PictureBox control.

To enable a PictureBox control to asynchronously load an image

1. Create an instance of the PictureBox component in your form.
2. Assign an event handler to the LoadCompleted event.

   Check for any errors that may have occurred during the asynchronous download here. This is also where you check for cancellation.

```
public Form1()
{
    InitializeComponent();

    this.pictureBox1.LoadCompleted +=
        new System.ComponentModel.AsyncCompletedEventHandler(this.pictureBox1_LoadCompleted);
}
```

```
Friend WithEvents PictureBox1 As System.Windows.Forms.PictureBox
```

```
private void pictureBox1_LoadCompleted(object sender, AsyncCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message, "Load Error");
    }
    else if (e.Cancelled)
    {
        MessageBox.Show("Load canceled", "Canceled");
    }
    else
    {
        MessageBox.Show("Load completed", "Completed");
    }
}
```

```
Private Sub PictureBox1_LoadCompleted( _
    ByVal sender As System.Object, _
    ByVal e As System.ComponentModel.AsyncCompletedEventArgs) _
    Handles PictureBox1.LoadCompleted

    If (e.Error IsNot Nothing) Then
        MessageBox.Show(e.Error.Message, "Load Error")
    ElseIf e.Cancelled Then
        MessageBox.Show("Load cancelled", "Canceled")
    Else
        MessageBox.Show("Load completed", "Completed")
    End If

End Sub
```

3. Add two buttons, called `loadButton` and `cancelLoadButton`, to your form. Add Click event handlers to start and cancel the download.

```
private void loadButton_Click(object sender, EventArgs e)
{
    // Replace with a real url.
    pictureBox1.LoadAsync("http://www.tailspintoys.com/image.jpg");
}
```

```
Private Sub loadButton_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles loadButton.Click

    ' Replace with a real url.
    PictureBox1.LoadAsync("http://www.tailspintoys.com/image.jpg")

End Sub
```

```
private void cancelLoadButton_Click(object sender, EventArgs e)
{
    pictureBox1.CancelAsync();
}
```

```
Private Sub cancelLoadButton_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles cancelLoadButton.Click

    PictureBox1.CancelAsync()

End Sub
```

4. Run your application.

   As the image download proceeds, you can move the form freely, minimize it, and maximize it.

See Also

How to: Run an Operation in the Background
Event-based Asynchronous Pattern Overview
NOT IN BUILD: Multithreading in Visual Basic

Asynchronous Programming Model (APM)

11/21/2017 • 4 min to read • Edit Online

An asynchronous operation that uses the IAsyncResult design pattern is implemented as two methods named **Begin** *OperationName* and **End** *OperationName* that begin and end the asynchronous operation *OperationName* respectively. For example, the FileStream class provides the BeginRead and EndRead methods to asynchronously read bytes from a file. These methods implement the asynchronous version of the Read method.

> **NOTE**
>
> Starting with the .NET Framework 4, the Task Parallel Library provides a new model for asynchronous and parallel programming. For more information, see Task Parallel Library (TPL) and Task-based Asynchronous Pattern (TAP)).

After calling **Begin** *OperationName*, an application can continue executing instructions on the calling thread while the asynchronous operation takes place on a different thread. For each call to **Begin** *OperationName*, the application should also call **End** *OperationName* to get the results of the operation.

Beginning an Asynchronous Operation

The **Begin** *OperationName* method begins asynchronous operation *OperationName* and returns an object that implements the IAsyncResult interface. IAsyncResult objects store information about an asynchronous operation. The following table shows information about an asynchronous operation.

MEMBER	DESCRIPTION
AsyncState	An optional application-specific object that contains information about the asynchronous operation.
AsyncWaitHandle	A WaitHandle that can be used to block application execution until the asynchronous operation completes.
CompletedSynchronously	A value that indicates whether the asynchronous operation completed on the thread used to call **Begin** *OperationName* instead of completing on a separate ThreadPool thread.
IsCompleted	A value that indicates whether the asynchronous operation has completed.

A **Begin** *OperationName* method takes any parameters declared in the signature of the synchronous version of the method that are passed by value or by reference. Any out parameters are not part of the **Begin** *OperationName* method signature. The **Begin** *OperationName* method signature also includes two additional parameters. The first of these defines an AsyncCallback delegate that references a method that is called when the asynchronous operation completes. The caller can specify `null` (`Nothing` in Visual Basic) if it does not want a method invoked when the operation completes. The second additional parameter is a user-defined object. This object can be used to pass application-specific state information to the method invoked when the asynchronous operation completes. If a **Begin** *OperationName* method takes additional operation-specific parameters, such as a byte array to store bytes read from a file, the AsyncCallback and application state object are the last parameters in the **Begin** *OperationName* method signature.

Begin *OperationName* returns control to the calling thread immediately. If the **Begin** *OperationName* method

throws exceptions, the exceptions are thrown before the asynchronous operation is started. If the **Begin** *OperationName* method throws exceptions, the callback method is not invoked.

Ending an Asynchronous Operation

The **End** *OperationName* method ends asynchronous operation *OperationName*. The return value of the **End** *OperationName* method is the same type returned by its synchronous counterpart and is specific to the asynchronous operation. For example, the EndRead method returns the number of bytes read from a FileStream and the EndGetHostByName method returns an IPHostEntry object that contains information about a host computer. The **End** *OperationName* method takes any out or ref parameters declared in the signature of the synchronous version of the method. In addition to the parameters from the synchronous method, the **End** *OperationName* method also includes an IAsyncResult parameter. Callers must pass the instance returned by the corresponding call to **Begin** *OperationName*.

If the asynchronous operation represented by the IAsyncResult object has not completed when **End** *OperationName* is called, **End** *OperationName* blocks the calling thread until the asynchronous operation is complete. Exceptions thrown by the asynchronous operation are thrown from the **End** *OperationName* method. The effect of calling the **End** *OperationName* method multiple times with the same IAsyncResult is not defined. Likewise, calling the **End** *OperationName* method with an IAsyncResult that was not returned by the related Begin method is also not defined.

> **NOTE**
> For either of the undefined scenarios, implementers should consider throwing InvalidOperationException.

> **NOTE**
> Implementers of this design pattern should notify the caller that the asynchronous operation completed by setting IsCompleted to true, calling the asynchronous callback method (if one was specified) and signaling the AsyncWaitHandle.

Application developers have several design choices for accessing the results of the asynchronous operation. The correct choice depends on whether the application has instructions that can execute while the operation completes. If an application cannot perform any additional work until it receives the results of the asynchronous operation, the application must block until the results are available. To block until an asynchronous operation completes, you can use one of the following approaches:

- Call **End** *OperationName* from the application's main thread, blocking application execution until the operation is complete. For an example that illustrates this technique, see Blocking Application Execution by Ending an Async Operation.
- Use the AsyncWaitHandle to block application execution until one or more operations are complete. For an example that illustrates this technique, see Blocking Application Execution Using an AsyncWaitHandle.

Applications that do not need to block while the asynchronous operation completes can use one of the following approaches:

- Poll for operation completion status by checking the IsCompleted property periodically and calling **End** *OperationName* when the operation is complete. For an example that illustrates this technique, see Polling for the Status of an Asynchronous Operation.
- Use an AsyncCallback delegate to specify a method to be invoked when the operation is complete. For an example that illustrates this technique, see Using an AsyncCallback Delegate to End an Asynchronous Operation.

See Also

Event-based Asynchronous Pattern (EAP)
Calling Synchronous Methods Asynchronously
Using an AsyncCallback Delegate and State Object

Calling Asynchronous Methods Using IAsyncResult

11/21/2017 • 1 min to read • Edit Online

Types in the .NET Framework and third-party class libraries can provide methods that allow an application to continue executing while performing asynchronous operations in threads other than the main application thread. The following sections describe and provide code examples that demonstrate the different ways you can call asynchronous methods that use the IAsyncResult design pattern.

- Blocking Application Execution by Ending an Async Operation.
- Blocking Application Execution Using an AsyncWaitHandle.
- Polling for the Status of an Asynchronous Operation.
- Using an AsyncCallback Delegate to End an Asynchronous Operation.

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Blocking Application Execution Using an AsyncWaitHandle

11/21/2017 • 3 min to read • Edit Online

Applications that cannot continue to do other work while waiting for the results of an asynchronous operation must block until the operation completes. Use one of the following options to block your application's main thread while waiting for an asynchronous operation to complete:

- Use the AsyncWaitHandle property of the IAsyncResult returned by the asynchronous operation's **Begin** *OperationName* method. This approach is demonstrated in this topic.
- Call the asynchronous operation's **End** *OperationName* method. For an example that demonstrates this approach, see Blocking Application Execution by Ending an Async Operation.

Applications that use one or more WaitHandle objects to block until an asynchronous operation is complete will typically call the **Begin** *OperationName* method, perform any work that can be done without the results of the operation, and then block until the asynchronous operation(s) completes. An application can block on a single operation by calling one of the WaitOne methods using the AsyncWaitHandle. To block while waiting for a set of asynchronous operations to complete, store the associated AsyncWaitHandle objects in an array and call one of the WaitAll methods. To block while waiting for any one of a set of asynchronous operations to complete, store the associated AsyncWaitHandle objects in an array and call one of the WaitAny methods.

Example

The following code example demonstrates using asynchronous methods in the DNS class to retrieve Domain Name System information for a user-specified computer. The example demonstrates blocking using the WaitHandle associated with the asynchronous operation. Note that `null` (`Nothing` in Visual Basic) is passed for the BeginGetHostByName `requestCallback` and `stateObject` parameters because these are not required when using this approach.

```
/*
The following example demonstrates using asynchronous methods to
get Domain Name System information for the specified host computer.

*/

using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class WaitUntilOperationCompletes
    {
        public static void Main(string[] args)
        {
            // Make sure the caller supplied a host name.
            if (args.Length == 0 || args[0].Length == 0)
            {
                // Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.");
                return;
            }
            // Start the asynchronous request for DNS information.
            IAsyncResult result = Dns.BeginGetHostEntry(args[0], null, null);
            Console.WriteLine("Processing request for information...");
            // Wait until the operation completes.
            result.AsyncWaitHandle.WaitOne();
            // The operation completed. Process the results.
            try
            {
                // Get the results.
                IPHostEntry host = Dns.EndGetHostEntry(result);
                string[] aliases = host.Aliases;
                IPAddress[] addresses = host.AddressList;
                if (aliases.Length > 0)
                {
                    Console.WriteLine("Aliases");
                    for (int i = 0; i < aliases.Length; i++)
                    {
                        Console.WriteLine("{0}", aliases[i]);
                    }
                }
                if (addresses.Length > 0)
                {
                    Console.WriteLine("Addresses");
                    for (int i = 0; i < addresses.Length; i++)
                    {
                        Console.WriteLine("{0}",addresses[i].ToString());
                    }
                }
            }
            catch (SocketException e)
            {
                Console.WriteLine("Exception occurred while processing the request: {0}",
                    e.Message);
            }
        }
    }
}
```

```
' The following example demonstrates using asynchronous methods to
' get Domain Name System information for the specified host computer.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading

namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class WaitUntilOperationCompletes

        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            Dim result as IAsyncResult= Dns.BeginGetHostEntry(args(0), Nothing, Nothing)
            Console.WriteLine("Processing request for information...")
            ' Wait until the operation completes.
            result.AsyncWaitHandle.WaitOne()
            ' The operation completed. Process the results.
            Try
                ' Get the results.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress= host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the request: {0}" _
                    , e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Blocking Application Execution by Ending an Async Operation

11/21/2017 • 2 min to read • Edit Online

Applications that cannot continue to do other work while waiting for the results of an asynchronous operation must block until the operation completes. Use one of the following options to block your application's main thread while waiting for an asynchronous operation to complete:

- Call the asynchronous operations **End** *OperationName* method. This approach is demonstrated in this topic.
- Use the AsyncWaitHandle property of the IAsyncResult returned by the asynchronous operation's **Begin** *OperationName* method. For an example that demonstrates this approach, see Blocking Application Execution Using an AsyncWaitHandle.

Applications that use the **End** *OperationName* method to block until an asynchronous operation is complete will typically call the **Begin** *OperationName* method, perform any work that can be done without the results of the operation, and then call **End** *OperationName*.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System information for a user-specified computer. Note that `null` (`Nothing` in Visual Basic) is passed for the BeginGetHostByName `requestCallback` and `stateObject` parameters because these arguments are not required when using this approach.

```
/*
The following example demonstrates using asynchronous methods to
get Domain Name System information for the specified host computer.
*/

using System;
using System.Net;
using System.Net.Sockets;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class BlockUntilOperationCompletes
    {
        public static void Main(string[] args)
        {
            // Make sure the caller supplied a host name.
            if (args.Length == 0 || args[0].Length == 0)
            {
                // Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.");
                return;
            }
            // Start the asynchronous request for DNS information.
            // This example does not use a delegate or user-supplied object
            // so the last two arguments are null.
            IAsyncResult result = Dns.BeginGetHostEntry(args[0], null, null);
            Console.WriteLine("Processing your request for information...");
            // Do any additional work that can be done here.
            try
            {
                // EndGetHostByName blocks until the process completes.
                IPHostEntry host = Dns.EndGetHostEntry(result);
                string[] aliases = host.Aliases;
                IPAddress[] addresses = host.AddressList;
                if (aliases.Length > 0)
                {
                    Console.WriteLine("Aliases");
                    for (int i = 0; i < aliases.Length; i++)
                    {
                        Console.WriteLine("{0}", aliases[i]);
                    }
                }
                if (addresses.Length > 0)
                {
                    Console.WriteLine("Addresses");
                    for (int i = 0; i < addresses.Length; i++)
                    {
                        Console.WriteLine("{0}",addresses[i].ToString());
                    }
                }
            }
            catch (SocketException e)
            {
                Console.WriteLine("An exception occurred while processing the request: {0}", e.Message);
            }
        }
    }
}
```

```
' The following example demonstrates using asynchronous methods to
' get Domain Name System information for the specified host computer.

Imports System
Imports System.Net
Imports System.Net.Sockets

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class BlockUntilOperationCompletes
        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            ' This example does not use a delegate or user-supplied object
            ' so the last two arguments are Nothing.
            Dim result as IAsyncResult = Dns.BeginGetHostEntry(args(0), Nothing, Nothing)
            Console.WriteLine("Processing your request for information...")
            ' Do any additional work that can be done here.
            Try
                ' EndGetHostByName blocks until the process completes.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress= host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the request: {0}", e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Polling for the Status of an Asynchronous Operation

11/21/2017 • 3 min to read • Edit Online

Applications that can do other work while waiting for the results of an asynchronous operation should not block waiting until the operation completes. Use one of the following options to continue executing instructions while waiting for an asynchronous operation to complete:

- Use the IsCompleted property of the IAsyncResult returned by the asynchronous operation's **Begin** *OperationName* method to determine whether the operation has completed. This approach is known as polling and is demonstrated in this topic.
- Use an AsyncCallback delegate to process the results of the asynchronous operation in a separate thread. For an example that demonstrates this approach, see Using an AsyncCallback Delegate to End an Asynchronous Operation.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System information for a user-specified computer. This example starts the asynchronous operation and then prints periods (".") at the console until the operation is complete. Note that **null** (**Nothing** in Visual Basic) is passed for the BeginGetHostByNameAsyncCallback and Object parameters because these arguments are not required when using this approach.

```
/*
The following example demonstrates using asynchronous methods to
get Domain Name System information for the specified host computer.
This example polls to detect the end of the asynchronous operation.
*/

using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class PollUntilOperationCompletes
    {
        static void UpdateUserInterface()
        {
            // Print a period to indicate that the application
            // is still working on the request.
            Console.Write(".");
        }
        public static void Main(string[] args)
        {
            // Make sure the caller supplied a host name.
            if (args.Length == 0 || args[0].Length == 0)
            {
                // Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.");
                return;
            }
            // Start the asychronous request for DNS information.
            IAsyncResult result = Dns.BeginGetHostEntry(args[0], null, null);
            Console.WriteLine("Processing request for information...");

            // Poll for completion information.
```

```
            // Print periods (".") until the operation completes.
            while (result.IsCompleted != true)
            {
                UpdateUserInterface();
            }
            // The operation is complete. Process the results.
            // Print a new line.
            Console.WriteLine();
            try
            {
                IPHostEntry host = Dns.EndGetHostEntry(result);
                string[] aliases = host.Aliases;
                IPAddress[] addresses = host.AddressList;
                if (aliases.Length > 0)
                {
                    Console.WriteLine("Aliases");
                    for (int i = 0; i < aliases.Length; i++)
                    {
                        Console.WriteLine("{0}", aliases[i]);
                    }
                }
                if (addresses.Length > 0)
                {
                    Console.WriteLine("Addresses");
                    for (int i = 0; i < addresses.Length; i++)
                    {
                        Console.WriteLine("{0}",addresses[i].ToString());
                    }
                }
            }
            catch (SocketException e)
            {
                Console.WriteLine("An exception occurred while processing the request: {0}", e.Message);
            }
        }
    }
}
```

```
'The following example demonstrates using asynchronous methods to
'get Domain Name System information for the specified host computer.
'This example polls to detect the end of the asynchronous operation.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class PollUntilOperationCompletes
        Shared Sub UpdateUserInterface()
            ' Print a period to indicate that the application
            ' is still working on the request.
            Console.Write(".")
        End Sub
        Public Shared Sub Main(args() as String)
            ' Make sure the caller supplied a host name.
            If(args.Length = 0)
                ' Print a message and exit.
                Console.WriteLine("You must specify the name of a host computer.")
                End
            End If
            ' Start the asynchronous request for DNS information.
            Dim result as IAsyncResult= Dns.BeginGetHostEntry(args(0), Nothing, Nothing)
            Console.WriteLine("Processing request for information...")

            ' Poll for completion information.
            ' Print periods (".") until the operation completes.
            Do while result.IsCompleted <> True
                UpdateUserInterface()
            Loop
            ' The operation is complete. Process the results.
            ' Print a new line.
            Console.WriteLine()
            Try
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                Dim  aliases() as String = host.Aliases
                Dim addresses() as IPAddress = host.AddressList
                Dim i as Integer
                If aliases.Length > 0
                    Console.WriteLine("Aliases")
                    For i = 0 To aliases.Length -1
                        Console.WriteLine("{0}", aliases(i))
                    Next i
                End If
                If addresses.Length > 0
                    Console.WriteLine("Addresses")
                    For i = 0 To addresses.Length -1
                        Console.WriteLine("{0}", addresses(i).ToString())
                    Next i
                End If
            Catch e as SocketException
                  Console.WriteLine("An exception occurred while processing the request: {0}", e.Message)
            End Try
        End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview

Using an AsyncCallback Delegate to End an Asynchronous Operation

11/21/2017 • 4 min to read • Edit Online

Applications that can do other work while waiting for the results of an asynchronous operation should not block waiting until the operation completes. Use one of the following options to continue executing instructions while waiting for an asynchronous operation to complete:

- Use an AsyncCallback delegate to process the results of the asynchronous operation in a separate thread. This approach is demonstrated in this topic.
- Use the IsCompleted property of the IAsyncResult returned by the asynchronous operation's **Begin** *OperationName* method to determine whether the operation has completed. For an example that demonstrates this approach, see Polling for the Status of an Asynchronous Operation.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System (DNS) information for user-specified computers. This example creates an AsyncCallback delegate that references the `ProcessDnsInformation` method. This method is called once for each asynchronous request for DNS information.

Note that the user-specified host is passed to the BeginGetHostByNameObject parameter. For an example that demonstrates defining and using a more complex state object, see Using an AsyncCallback Delegate and State Object.

```
/*
The following example demonstrates using asynchronous methods to
get Domain Name System information for the specified host computers.
This example uses a delegate to obtain the results of each asynchronous
operation.
*/

using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Collections.Specialized;
using System.Collections;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class UseDelegateForAsyncCallback
    {
        static int requestCounter;
        static ArrayList hostData = new ArrayList();
        static StringCollection hostNames = new StringCollection();
        static void UpdateUserInterface()
        {
            // Print a message to indicate that the application
            // is still working on the remaining requests.
            Console.WriteLine("{0} requests remaining.", requestCounter);
        }
        public static void Main()
        {
            // Create the delegate that will process the results of the
            // asynchronous request
```

```
        // asynchronous request.
        AsyncCallback callBack = new AsyncCallback(ProcessDnsInformation);
        string host;
        do
        {
            Console.Write(" Enter the name of a host computer or <enter> to finish: ");
            host = Console.ReadLine();
            if (host.Length > 0)
            {
                // Increment the request counter in a thread safe manner.
                Interlocked.Increment(ref requestCounter);
                // Start the asynchronous request for DNS information.
                Dns.BeginGetHostEntry(host, callBack, host);
             }
        } while (host.Length > 0);
        // The user has entered all of the host names for lookup.
        // Now wait until the threads complete.
        while (requestCounter > 0)
        {
            UpdateUserInterface();
        }
        // Display the results.
        for (int i = 0; i< hostNames.Count; i++)
        {
            object data = hostData [i];
            string message = data as string;
            // A SocketException was thrown.
            if (message != null)
            {
                Console.WriteLine("Request for {0} returned message: {1}",
                    hostNames[i], message);
                continue;
            }
            // Get the results.
            IPHostEntry h = (IPHostEntry) data;
            string[] aliases = h.Aliases;
            IPAddress[] addresses = h.AddressList;
            if (aliases.Length > 0)
            {
                Console.WriteLine("Aliases for {0}", hostNames[i]);
                for (int j = 0; j < aliases.Length; j++)
                {
                    Console.WriteLine("{0}", aliases[j]);
                }
            }
            if (addresses.Length > 0)
            {
                Console.WriteLine("Addresses for {0}", hostNames[i]);
                for (int k = 0; k < addresses.Length; k++)
                {
                    Console.WriteLine("{0}",addresses[k].ToString());
                }
            }
        }
    }

     // The following method is called when each asynchronous operation completes.
     static void ProcessDnsInformation(IAsyncResult result)
     {
         string hostName = (string) result.AsyncState;
         hostNames.Add(hostName);
         try
         {
             // Get the results.
             IPHostEntry host = Dns.EndGetHostEntry(result);
             hostData.Add(host);
         }
         // Store the exception message.
         catch (SocketException e)
```

```
            {
                hostData.Add(e.Message);
            }
            finally
            {
                // Decrement the request counter in a thread-safe manner.
                Interlocked.Decrement(ref requestCounter);
            }
        }
    }
}
```

```
'The following example demonstrates using asynchronous methods to
'get Domain Name System information for the specified host computers.
'This example uses a delegate to obtain the results of each asynchronous
'operation.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading
Imports System.Collections.Specialized
Imports System.Collections

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class UseDelegateForAsyncCallback

        Dim Shared requestCounter as Integer
        Dim Shared hostData as ArrayList = new ArrayList()
        Dim Shared hostNames as StringCollection = new StringCollection()
        Shared Sub UpdateUserInterface()

            ' Print a message to indicate that the application
            ' is still working on the remaining requests.
            Console.WriteLine("{0} requests remaining.", requestCounter)
        End Sub
        Public Shared Sub Main()
            ' Create the delegate that will process the results of the
            ' asynchronous request.
            Dim callBack as AsyncCallback
            Dim host as string
            Dim i, j, k as Integer
            callback = AddressOf ProcessDnsInformation
            Do
                Console.Write(" Enter the name of a host computer or <enter> to finish: ")
                host = Console.ReadLine()
                If host.Length > 0
                    ' Increment the request counter in a thread safe manner.
                    Interlocked.Increment(requestCounter)
                    ' Start the asynchronous request for DNS information.
                    Dns.BeginGetHostEntry(host, callBack, host)
                End If
            Loop While (host.Length > 0)

            ' The user has entered all of the host names for lookup.
            ' Now wait until the threads complete.
            Do While requestCounter > 0

                UpdateUserInterface()
            Loop

            ' Display the results.
            For i = 0 To hostNames.Count -1
                Dim dataObject as Object = hostData (i)
                Dim message as String
```

```
                ' Was a SocketException was thrown?
                If TypeOf dataObject is String
                    message = CType(dataObject, String)
                    Console.WriteLine("Request for {0} returned message: {1}", _
                        hostNames(i), message)
                Else
                    ' Get the results.
                    Dim h as IPHostEntry = CType(dataObject, IPHostEntry)
                    Dim aliases() as String = h.Aliases
                    Dim addresses() as IPAddress = h.AddressList
                    If aliases.Length > 0
                        Console.WriteLine("Aliases for 0}", hostNames(i))
                        For j = 0 To aliases.Length -1
                            Console.WriteLine("{0}", aliases(j))
                        Next j
                    End If
                    If addresses.Length > 0
                        Console.WriteLine("Addresses for {0}", hostNames(i))
                        For k = 0 To addresses.Length -1
                            Console.WriteLine("{0}",addresses(k).ToString())
                        Next k
                    End If
                End If
            Next i
        End Sub

         ' The following method is called when each asynchronous operation completes.
         Shared Sub ProcessDnsInformation(result as IAsyncResult)

            Dim hostName as String = CType(result.AsyncState, String)
            hostNames.Add(hostName)
            Try
                ' Get the results.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                hostData.Add(host)
            ' Store the exception message.
            Catch e as SocketException
                hostData.Add(e.Message)
            Finally
                ' Decrement the request counter in a thread-safe manner.
                Interlocked.Decrement(requestCounter)
            End Try
         End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview
Calling Asynchronous Methods Using IAsyncResult
Using an AsyncCallback Delegate and State Object

Using an AsyncCallback Delegate and State Object

11/21/2017 • 5 min to read • Edit Online

When you use an AsyncCallback delegate to process the results of the asynchronous operation in a separate thread, you can use a state object to pass information between the callbacks and to retrieve a final result. This topic demonstrates that practice by expanding the example in Using an AsyncCallback Delegate to End an Asynchronous Operation.

Example

The following code example demonstrates using asynchronous methods in the Dns class to retrieve Domain Name System (DNS) information for user-specified computers. This example defines and uses the `HostRequest` class to store state information. A `HostRequest` object gets created for each computer name entered by the user. This object is passed to the BeginGetHostByName method. The `ProcessDnsInformation` method is called each time a request completes. The `HostRequest` object is retrieved using the AsyncState property. The `ProcessDnsInformation` method uses the `HostRequest` object to store the IPHostEntry returned by the request or a SocketException thrown by the request. When all requests are complete, the application iterates over the `HostRequest` objects and displays the DNS information or SocketException error message.

```
/*
The following example demonstrates using asynchronous methods to
get Domain Name System information for the specified host computer.
*/

using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Collections;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
// Create a state object that holds each requested host name,
// an associated IPHostEntry object or a SocketException.
    public class HostRequest
    {
        // Stores the requested host name.
        private string hostName;
        // Stores any SocketException returned by the Dns EndGetHostByName method.
        private SocketException e;
        // Stores an IPHostEntry returned by the Dns EndGetHostByName method.
        private IPHostEntry entry;

        public HostRequest(string name)
        {
            hostName = name;
        }

        public string HostName
        {
            get
            {
                return hostName;
            }
        }

        public SocketException ExceptionObject
        {
```

```
        {
            get
            {
                return e;
            }
            set
            {
                 e = value;
            }
        }

        public IPHostEntry HostEntry
        {
            get
            {
                return entry;
            }
            set
            {
                entry = value;
            }
        }
    }

    public class UseDelegateAndStateForAsyncCallback
    {
        // The number of pending requests.
        static int requestCounter;
        static ArrayList hostData = new ArrayList();
        static void UpdateUserInterface()
        {
            // Print a message to indicate that the application
            // is still working on the remaining requests.
            Console.WriteLine("{0} requests remaining.", requestCounter);
        }
        public static void Main()
        {
            // Create the delegate that will process the results of the
            // asynchronous request.
            AsyncCallback callBack = new AsyncCallback(ProcessDnsInformation);
            string host;
            do
            {
                Console.Write(" Enter the name of a host computer or <enter> to finish: ");
                host = Console.ReadLine();
                if (host.Length > 0)
                {
                    // Increment the request counter in a thread safe manner.
                    Interlocked.Increment(ref requestCounter);
                    // Create and store the state object for this request.
                    HostRequest request = new HostRequest(host);
                    hostData.Add(request);
                    // Start the asynchronous request for DNS information.
                    Dns.BeginGetHostEntry(host, callBack, request);
                 }
            } while (host.Length > 0);
            // The user has entered all of the host names for lookup.
            // Now wait until the threads complete.
            while (requestCounter > 0)
            {
                UpdateUserInterface();
            }
            // Display the results.
            foreach(HostRequest r in hostData)
            {
                    if (r.ExceptionObject != null)
                    {
                        Console.WriteLine("Request for host {0} returned the following error: {1}.",
                            r.HostName, r.ExceptionObject.Message);
                    }
```

```
                    }
                    else
                    {
                        // Get the results.
                        IPHostEntry h = r.HostEntry;
                        string[] aliases = h.Aliases;
                        IPAddress[] addresses = h.AddressList;
                        if (aliases.Length > 0)
                        {
                            Console.WriteLine("Aliases for {0}", r.HostName);
                            for (int j = 0; j < aliases.Length; j++)
                            {
                                Console.WriteLine("{0}", aliases[j]);
                            }
                        }
                        if (addresses.Length > 0)
                        {
                            Console.WriteLine("Addresses for {0}", r.HostName);
                            for (int k = 0; k < addresses.Length; k++)
                            {
                                Console.WriteLine("{0}",addresses[k].ToString());
                            }
                        }
                    }
                }
            }

            // The following method is invoked when each asynchronous operation completes.
            static void ProcessDnsInformation(IAsyncResult result)
            {
               // Get the state object associated with this request.
               HostRequest request = (HostRequest) result.AsyncState;
                try
                {
                    // Get the results and store them in the state object.
                    IPHostEntry host = Dns.EndGetHostEntry(result);
                    request.HostEntry = host;
                }
                catch (SocketException e)
                {
                    // Store any SocketExceptions.
                    request.ExceptionObject = e;
                }
                finally
                {
                    // Decrement the request counter in a thread-safe manner.
                    Interlocked.Decrement(ref requestCounter);
                }
            }
        }
    }
```

```
' The following example demonstrates using asynchronous methods to
' get Domain Name System information for the specified host computer.

Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading
Imports System.Collections

Namespace Examples.AdvancedProgramming.AsynchronousOperations
' Create a state object that holds each requested host name,
' an associated IPHostEntry object or a SocketException.
    Public Class HostRequest
        ' Stores the requested host name.
        Dim hostNameValue as string
        ' Stores any SocketException returned by the Dns.EndGetHostByName method.
```

```
    ' Stores any SocketException returned by the Dns EndGetHostByName method.
    Dim e as SocketException
    ' Stores an IPHostEntry returned by the Dns EndGetHostByName method.
    Dim entry as IPHostEntry

    Public Sub New(name as String)
        hostNameValue = name
    End Sub

    ReadOnly Public Property HostName as String
        Get
            return hostNameValue
        End Get
    End Property

    Public Property ExceptionObject as SocketException
        Get
            return e
        End Get
        Set
             e = value
        End Set
    End Property

    Public Property HostEntry as IPHostEntry
        Get
            return entry
        End Get
        Set
            entry = value
        End Set
    End Property
End Class

Public Class UseDelegateAndStateForAsyncCallback
    ' The number of pending requests.
    Dim Shared requestCounter as Integer
    Dim Shared hostData as ArrayList = new ArrayList()

    Shared Sub UpdateUserInterface()
        ' Print a message to indicate that the application
        ' is still working on the remaining requests.
        Console.WriteLine("{0} requests remaining.", requestCounter)
    End Sub

    Public Shared Sub Main()
        ' Create the delegate that will process the results of the
        ' asynchronous request.
        Dim callBack as AsyncCallback = AddressOf ProcessDnsInformation
        Dim host as string

        Do
            Console.Write(" Enter the name of a host computer or <enter> to finish: ")
            host = Console.ReadLine()
            If host.Length > 0
                ' Increment the request counter in a thread safe manner.
                Interlocked.Increment (requestCounter)
                ' Create and store the state object for this request.
                Dim  request as HostRequest = new HostRequest(host)
                hostData.Add(request)
                ' Start the asynchronous request for DNS information.
                Dns.BeginGetHostEntry(host, callBack, request)
             End If
        Loop While host.Length > 0

        ' The user has entered all of the host names for lookup.
        ' Now wait until the threads complete.
        Do While requestCounter > 0
            UpdateUserInterface()
        Loop
```

```
            Loop

            ' Display the results.
            For Each r as HostRequest In hostData
                    If IsNothing(r.ExceptionObject) = False
                        Console.WriteLine( _
                          "Request for host {0} returned the following error: {1}.", _
                          r.HostName, r.ExceptionObject.Message)
                    Else
                        ' Get the results.
                        Dim h as IPHostEntry = r.HostEntry
                        Dim aliases() as String = h.Aliases
                        Dim addresses() as IPAddress = h.AddressList
                        Dim j, k as Integer

                        If aliases.Length > 0
                            Console.WriteLine("Aliases for {0}", r.HostName)
                            For j = 0 To aliases.Length - 1
                                Console.WriteLine("{0}", aliases(j))
                            Next j
                        End If
                        If addresses.Length > 0
                            Console.WriteLine("Addresses for {0}", r.HostName)
                            For k = 0 To addresses.Length -1
                                Console.WriteLine("{0}",addresses(k).ToString())
                            Next k
                        End If
                    End If
            Next r
        End Sub

         ' The following method is invoked when each asynchronous operation completes.
         Shared Sub ProcessDnsInformation(result as IAsyncResult)
            ' Get the state object associated with this request.
            Dim request as HostRequest= CType(result.AsyncState,HostRequest)
             Try
                 ' Get the results and store them in the state object.
                Dim host as IPHostEntry = Dns.EndGetHostEntry(result)
                request.HostEntry = host
             Catch e as SocketException
                 ' Store any SocketExceptions.
                 request.ExceptionObject = e
             Finally
                 ' Decrement the request counter in a thread-safe manner.
                 Interlocked.Decrement(requestCounter)
             End Try
         End Sub
    End Class
End Namespace
```

See Also

Event-based Asynchronous Pattern (EAP)
Event-based Asynchronous Pattern Overview
Using an AsyncCallback Delegate to End an Asynchronous Operation

Asynchronous Programming Using Delegates

11/11/2017 • 1 min to read • Edit Online

Delegates enable you to call a synchronous method in an asynchronous manner. When you call a delegate synchronously, the `Invoke` method calls the target method directly on the current thread. If the `BeginInvoke` method is called, the common language runtime (CLR) queues the request and returns immediately to the caller. The target method is called asynchronously on a thread from the thread pool. The original thread, which submitted the request, is free to continue executing in parallel with the target method. If a callback method has been specified in the call to the `BeginInvoke` method, the callback method is called when the target method ends. In the callback method, the `EndInvoke` method obtains the return value and any input/output or output-only parameters. If no callback method is specified when calling `BeginInvoke`, `EndInvoke` can be called from the thread that called `BeginInvoke`.

IMPORTANT

Compilers should emit delegate classes with `Invoke`, `BeginInvoke`, and `EndInvoke` methods using the delegate signature specified by the user. The `BeginInvoke` and `EndInvoke` methods should be decorated as native. Because these methods are marked as native, the CLR automatically provides the implementation at class load time. The loader ensures that they are not overridden.

In This Section

Calling Synchronous Methods Asynchronously

Discusses the use of delegates to make asynchronous calls to ordinary methods, and provides simple code examples that show the four ways to wait for an asynchronous call to return.

Related Sections

Event-based Asynchronous Pattern (EAP)

Describes asynchronous programming with the .NET Framework.

See Also

Delegate

Calling Synchronous Methods Asynchronously

11/21/2017 • 18 min to read • Edit Online

The .NET Framework enables you to call any method asynchronously. To do this you define a delegate with the same signature as the method you want to call; the common language runtime automatically defines `BeginInvoke` and `EndInvoke` methods for this delegate, with the appropriate signatures.

> **NOTE**
>
> Asynchronous delegate calls, specifically the `BeginInvoke` and `EndInvoke` methods, are not supported in the .NET Compact Framework.

The `BeginInvoke` method initiates the asynchronous call. It has the same parameters as the method that you want to execute asynchronously, plus two additional optional parameters. The first parameter is an AsyncCallback delegate that references a method to be called when the asynchronous call completes. The second parameter is a user-defined object that passes information into the callback method. `BeginInvoke` returns immediately and does not wait for the asynchronous call to complete. `BeginInvoke` returns an IAsyncResult, which can be used to monitor the progress of the asynchronous call.

The `EndInvoke` method retrieves the results of the asynchronous call. It can be called any time after `BeginInvoke`. If the asynchronous call has not completed, `EndInvoke` blocks the calling thread until it completes. The parameters of `EndInvoke` include the `out` and `ref` parameters (`<Out>` `ByRef` and `ByRef` in Visual Basic) of the method that you want to execute asynchronously, plus the IAsyncResult returned by `BeginInvoke`.

> **NOTE**
>
> The IntelliSense feature in Visual Studio 2005 displays the parameters of `BeginInvoke` and `EndInvoke`. If you are not using Visual Studio or a similar tool, or if you are using C# with Visual Studio 2005, see Asynchronous Programming Model (APM) for a description of the parameters defined for these methods.

The code examples in this topic demonstrate four common ways to use `BeginInvoke` and `EndInvoke` to make asynchronous calls. After calling `BeginInvoke` you can do the following:

- Do some work and then call `EndInvoke` to block until the call completes.
- Obtain a WaitHandle using the IAsyncResult.AsyncWaitHandle property, use its WaitOne method to block execution until the WaitHandle is signaled, and then call `EndInvoke`.
- Poll the IAsyncResult returned by `BeginInvoke` to determine when the asynchronous call has completed, and then call `EndInvoke`.
- Pass a delegate for a callback method to `BeginInvoke`. The method is executed on a ThreadPool thread when the asynchronous call completes. The callback method calls `EndInvoke`.

> **IMPORTANT**
>
> No matter which technique you use, always call `EndInvoke` to complete your asynchronous call.

Defining the Test Method and Asynchronous Delegate

The code examples that follow demonstrate various ways of calling the same long-running method, `TestMethod`, asynchronously. The `TestMethod` method displays a console message to show that it has begun processing, sleeps for a few seconds, and then ends. `TestMethod` has an `out` parameter to demonstrate the way such parameters are added to the signatures of `BeginInvoke` and `EndInvoke`. You can handle `ref` parameters similarly.

The following code example shows the definition of `TestMethod` and the delegate named `AsyncMethodCaller` that can be used to call `TestMethod` asynchronously. To compile the code examples, you must include the definitions for `TestMethod` and the `AsyncMethodCaller` delegate.

```
using namespace System;
using namespace System::Threading;
using namespace System::Runtime::InteropServices;

namespace Examples {
namespace AdvancedProgramming {
namespace AsynchronousOperations
{
    public ref class AsyncDemo
    {
    public:
        // The method to be executed asynchronously.
        String^ TestMethod(int callDuration, [OutAttribute] int% threadId)
        {
            Console::WriteLine("Test method begins.");
            Thread::Sleep(callDuration);
            threadId = Thread::CurrentThread->ManagedThreadId;
            return String::Format("My call time was {0}.", callDuration);
        }
    };

    // The delegate must have the same signature as the method
    // it will call asynchronously.
    public delegate String^ AsyncMethodCaller(int callDuration, [OutAttribute] int% threadId);
}}}
```

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncDemo
    {
        // The method to be executed asynchronously.
        public string TestMethod(int callDuration, out int threadId)
        {
            Console.WriteLine("Test method begins.");
            Thread.Sleep(callDuration);
            threadId = Thread.CurrentThread.ManagedThreadId;
            return String.Format("My call time was {0}.", callDuration.ToString());
        }
    }
    // The delegate must have the same signature as the method
    // it will call asynchronously.
    public delegate string AsyncMethodCaller(int callDuration, out int threadId);
}
```

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class AsyncDemo
        ' The method to be executed asynchronously.
        Public Function TestMethod(ByVal callDuration As Integer, _
                <Out> ByRef threadId As Integer) As String
            Console.WriteLine("Test method begins.")
            Thread.Sleep(callDuration)
            threadId = Thread.CurrentThread.ManagedThreadId()
            return String.Format("My call time was {0}.", callDuration.ToString())
        End Function
    End Class

    ' The delegate must have the same signature as the method
    ' it will call asynchronously.
    Public Delegate Function AsyncMethodCaller(ByVal callDuration As Integer, _
        <Out> ByRef threadId As Integer) As String
End Namespace
```

Waiting for an Asynchronous Call with EndInvoke

The simplest way to execute a method asynchronously is to start executing the method by calling the delegate's `BeginInvoke` method, do some work on the main thread, and then call the delegate's `EndInvoke` method. `EndInvoke` might block the calling thread because it does not return until the asynchronous call completes. This is a good technique to use with file or network operations.

> **IMPORTANT**
>
> Because `EndInvoke` might block, you should never call it from threads that service the user interface.

```
#using <TestMethod.dll>

using namespace System;
using namespace System::Threading;
using namespace Examples::AdvancedProgramming::AsynchronousOperations;

void main()
{
    // The asynchronous method puts the thread id here.
    int threadId = 2546;

    // Create an instance of the test class.
    AsyncDemo^ ad = gcnew AsyncDemo();

    // Create the delegate.
    AsyncMethodCaller^ caller = gcnew AsyncMethodCaller(ad, &AsyncDemo::TestMethod);

    // Initiate the asychronous call.
    IAsyncResult^ result = caller->BeginInvoke(3000,
        threadId, nullptr, nullptr);

    Thread::Sleep(1);
    Console::WriteLine("Main thread {0} does some work.",
        Thread::CurrentThread->ManagedThreadId);

    // Call EndInvoke to wait for the asynchronous call to complete,
    // and to retrieve the results.
    String^ returnValue = caller->EndInvoke(threadId, result);

    Console::WriteLine("The call executed on thread {0}, with return value \"{1}\".",
        threadId, returnValue);
}

/* This example produces output similar to the following:

Main thread 1 does some work.
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncMain
    {
        public static void Main()
        {
            // The asynchronous method puts the thread id here.
            int threadId;

            // Create an instance of the test class.
            AsyncDemo ad = new AsyncDemo();

            // Create the delegate.
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            // Initiate the asychronous call.
            IAsyncResult result = caller.BeginInvoke(3000,
                out threadId, null, null);

            Thread.Sleep(0);
            Console.WriteLine("Main thread {0} does some work.",
                Thread.CurrentThread.ManagedThreadId);

            // Call EndInvoke to wait for the asynchronous call to complete,
            // and to retrieve the results.
            string returnValue = caller.EndInvoke(out threadId, result);

            Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".",
                threadId, returnValue);
        }
    }
}

/* This example produces output similar to the following:

Main thread 1 does some work.
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations
    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            Thread.Sleep(0)
            Console.WriteLine("Main thread {0} does some work.", _
                 Thread.CurrentThread.ManagedThreadId)

            ' Call EndInvoke to Wait for the asynchronous call to complete,
            ' and to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            Console.WriteLine("The call executed on thread {0}, with return value ""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class

End Namespace

'This example produces output similar to the following:
'
'Main thread 1 does some work.
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
```

Waiting for an Asynchronous Call with WaitHandle

You can obtain a WaitHandle by using the AsyncWaitHandle property of the IAsyncResult returned by `BeginInvoke`. The WaitHandle is signaled when the asynchronous call completes, and you can wait for it by calling the WaitOne method.

If you use a WaitHandle, you can perform additional processing before or after the asynchronous call completes, but before calling `EndInvoke` to retrieve the results.

> **NOTE**
>
> The wait handle is not closed automatically when you call `EndInvoke`. If you release all references to the wait handle, system resources are freed when garbage collection reclaims the wait handle. To free the system resources as soon as you are finished using the wait handle, dispose of it by calling the WaitHandle.Close method. Garbage collection works more efficiently when disposable objects are explicitly disposed.

```
#using <TestMethod.dll>

using namespace System;
using namespace System::Threading;
using namespace Examples::AdvancedProgramming::AsynchronousOperations;

void main()
{
    // The asynchronous method puts the thread id here.
    int threadId;

    // Create an instance of the test class.
    AsyncDemo^ ad = gcnew AsyncDemo();

    // Create the delegate.
    AsyncMethodCaller^ caller = gcnew AsyncMethodCaller(ad, &AsyncDemo::TestMethod);

    // Initiate the asychronous call.
    IAsyncResult^ result = caller->BeginInvoke(3000,
        threadId, nullptr, nullptr);

    Thread::Sleep(0);
    Console::WriteLine("Main thread {0} does some work.",
        Thread::CurrentThread->ManagedThreadId);

    // Wait for the WaitHandle to become signaled.
    result->AsyncWaitHandle->WaitOne();

    // Perform additional processing here.
    // Call EndInvoke to retrieve the results.
    String^ returnValue = caller->EndInvoke(threadId, result);

    // Close the wait handle.
    result->AsyncWaitHandle->Close();

    Console::WriteLine("The call executed on thread {0}, with return value \"{1}\".",
        threadId, returnValue);
}

/* This example produces output similar to the following:

Main thread 1 does some work.
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncMain
    {
        static void Main()
        {
            // The asynchronous method puts the thread id here.
            int threadId;

            // Create an instance of the test class.
            AsyncDemo ad = new AsyncDemo();

            // Create the delegate.
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            // Initiate the asychronous call.
            IAsyncResult result = caller.BeginInvoke(3000,
                out threadId, null, null);

            Thread.Sleep(0);
            Console.WriteLine("Main thread {0} does some work.",
                Thread.CurrentThread.ManagedThreadId);

            // Wait for the WaitHandle to become signaled.
            result.AsyncWaitHandle.WaitOne();

            // Perform additional processing here.
            // Call EndInvoke to retrieve the results.
            string returnValue = caller.EndInvoke(out threadId, result);

            // Close the wait handle.
            result.AsyncWaitHandle.Close();

            Console.WriteLine("The call executed on thread {0}, with return value \"{1}\".",
                threadId, returnValue);
        }
    }
}

/* This example produces output similar to the following:

Main thread 1 does some work.
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            Thread.Sleep(0)
            Console.WriteLine("Main thread {0} does some work.", _
                Thread.CurrentThread.ManagedThreadId)
            ' Perform additional processing here and then
            ' wait for the WaitHandle to be signaled.
            result.AsyncWaitHandle.WaitOne()

            ' Call EndInvoke to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            ' Close the wait handle.
            result.AsyncWaitHandle.Close()

            Console.WriteLine("The call executed on thread {0}, with return value ""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class
End Namespace

'This example produces output similar to the following:
'
'Main thread 1 does some work.
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
```

Polling for Asynchronous Call Completion

You can use the IsCompleted property of the IAsyncResult returned by `BeginInvoke` to discover when the asynchronous call completes. You might do this when making the asynchronous call from a thread that services the user interface. Polling for completion allows the calling thread to continue executing while the asynchronous call executes on a ThreadPool thread.

```
#using <TestMethod.dll>

using namespace System;
using namespace System::Threading;
using namespace Examples::AdvancedProgramming::AsynchronousOperations;

void main()
{
    // The asynchronous method puts the thread id here.
    int threadId;

    // Create an instance of the test class.
    AsyncDemo^ ad = gcnew AsyncDemo();

    // Create the delegate.
    AsyncMethodCaller^ caller = gcnew AsyncMethodCaller(ad, &AsyncDemo::TestMethod);

    // Initiate the asychronous call.
    IAsyncResult^ result = caller->BeginInvoke(3000,
        threadId, nullptr, nullptr);

    // Poll while simulating work.
    while(result->IsCompleted == false)
    {
        Thread::Sleep(250);
        Console::Write(".");
    }

    // Call EndInvoke to retrieve the results.
    String^ returnValue = caller->EndInvoke(threadId, result);

    Console::WriteLine("\nThe call executed on thread {0}, with return value \"{1}\".",
        threadId, returnValue);
}

/* This example produces output similar to the following:

Test method begins.
.............
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
using System;
using System.Threading;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncMain
    {
        static void Main() {
            // The asynchronous method puts the thread id here.
            int threadId;

            // Create an instance of the test class.
            AsyncDemo ad = new AsyncDemo();

            // Create the delegate.
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            // Initiate the asychronous call.
            IAsyncResult result = caller.BeginInvoke(3000,
                out threadId, null, null);

            // Poll while simulating work.
            while(result.IsCompleted == false) {
                Thread.Sleep(250);
                Console.Write(".");
            }

            // Call EndInvoke to retrieve the results.
            string returnValue = caller.EndInvoke(out threadId, result);

            Console.WriteLine("\nThe call executed on thread {0}, with return value \"{1}\".",
                threadId, returnValue);
        }
    }
}

/* This example produces output similar to the following:

Test method begins.
.............
The call executed on thread 3, with return value "My call time was 3000.".
 */
```

```
Imports System
Imports System.Threading
Imports System.Runtime.InteropServices

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class AsyncMain
        Shared Sub Main()
            ' The asynchronous method puts the thread id here.
            Dim threadId As Integer

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' Initiate the asynchronous call.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                threadId, Nothing, Nothing)

            ' Poll while simulating work.
            While result.IsCompleted = False
                Thread.Sleep(250)
                Console.Write(".")
            End While

            ' Call EndInvoke to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, result)

            Console.WriteLine(vbCrLf & _
                "The call executed on thread {0}, with return value ""{1}"".", _
                threadId, returnValue)
        End Sub
    End Class
End Namespace

' This example produces output similar to the following:
'
'Test method begins.
'.............
'The call executed on thread 3, with return value "My call time was 3000.".
```

Executing a Callback Method When an Asynchronous Call Completes

If the thread that initiates the asynchronous call does not need to be the thread that processes the results, you can execute a callback method when the call completes. The callback method is executed on a ThreadPool thread.

To use a callback method, you must pass `BeginInvoke` an AsyncCallback delegate that represents the callback method. You can also pass an object that contains information to be used by the callback method. In the callback method, you can cast the IAsyncResult, which is the only parameter of the callback method, to an AsyncResult object. You can then use the AsyncResult.AsyncDelegate property to get the delegate that was used to initiate the call so that you can call `EndInvoke`.

Notes on the example:

- The `threadId` parameter of `TestMethod` is an `out` parameter ([`<Out>` `ByRef` in Visual Basic), so its input value is never used by `TestMethod`. A dummy variable is passed to the `BeginInvoke` call. If the `threadId` parameter were a `ref` parameter (`ByRef` in Visual Basic), the variable would have to be a class-level field so that it could be passed to both `BeginInvoke` and `EndInvoke`.
- The state information that is passed to `BeginInvoke` is a format string, which the callback method uses to

format an output message. Because it is passed as type Object, the state information must be cast to its proper type before it can be used.

- The callback is made on a ThreadPool thread. ThreadPool threads are background threads, which do not keep the application running if the main thread ends, so the main thread of the example has to sleep long enough for the callback to finish.

```
#using <TestMethod.dll>

using namespace System;
using namespace System::Threading;
using namespace System::Runtime::Remoting::Messaging;
using namespace Examples::AdvancedProgramming::AsynchronousOperations;

// The callback method must have the same signature as the
// AsyncCallback delegate.
void CallbackMethod(IAsyncResult^ ar)
{
    // Retrieve the delegate.
    AsyncResult^ result = (AsyncResult^) ar;
    AsyncMethodCaller^ caller = (AsyncMethodCaller^) result->AsyncDelegate;

    // Retrieve the format string that was passed as state
    // information.
    String^ formatString = (String^) ar->AsyncState;

    // Define a variable to receive the value of the out parameter.
    // If the parameter were ref rather than out then it would have to
    // be a class-level field so it could also be passed to BeginInvoke.
    int threadId = 0;

    // Call EndInvoke to retrieve the results.
    String^ returnValue = caller->EndInvoke(threadId, ar);

    // Use the format string to format the output message.
    Console::WriteLine(formatString, threadId, returnValue);
};

void main()
{
    // Create an instance of the test class.
    AsyncDemo^ ad = gcnew AsyncDemo();

    // Create the delegate.
    AsyncMethodCaller^ caller = gcnew AsyncMethodCaller(ad, &AsyncDemo::TestMethod);

    // The threadId parameter of TestMethod is an out parameter, so
    // its input value is never used by TestMethod. Therefore, a dummy
    // variable can be passed to the BeginInvoke call. If the threadId
    // parameter were a ref parameter, it would have to be a class-
    // level field so that it could be passed to both BeginInvoke and
    // EndInvoke.
    int dummy = 0;

    // Initiate the asynchronous call, passing three seconds (3000 ms)
    // for the callDuration parameter of TestMethod; a dummy variable
    // for the out parameter (threadId); the callback delegate; and
    // state information that can be retrieved by the callback method.
    // In this case, the state information is a string that can be used
    // to format a console message.
    IAsyncResult^ result = caller->BeginInvoke(3000,
        dummy,
        gcnew AsyncCallback(&CallbackMethod),
        "The call executed on thread {0}, with return value \"{1}\".");

    Console::WriteLine("The main thread {0} continues to execute...",
        Thread::CurrentThread->ManagedThreadId);
```

```
        Thread.CurrentThread.ManagedThreadId);

    // The callback is made on a ThreadPool thread. ThreadPool threads
    // are background threads, which do not keep the application running
    // if the main thread ends. Comment out the next line to demonstrate
    // this.
    Thread::Sleep(4000);
    Console::WriteLine("The main thread ends.");
}

/* This example produces output similar to the following:

The main thread 1 continues to execute...
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
The main thread ends.
 */
```

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace Examples.AdvancedProgramming.AsynchronousOperations
{
    public class AsyncMain
    {
        static void Main()
        {
            // Create an instance of the test class.
            AsyncDemo ad = new AsyncDemo();

            // Create the delegate.
            AsyncMethodCaller caller = new AsyncMethodCaller(ad.TestMethod);

            // The threadId parameter of TestMethod is an out parameter, so
            // its input value is never used by TestMethod. Therefore, a dummy
            // variable can be passed to the BeginInvoke call. If the threadId
            // parameter were a ref parameter, it would have to be a class-
            // level field so that it could be passed to both BeginInvoke and
            // EndInvoke.
            int dummy = 0;

            // Initiate the asynchronous call, passing three seconds (3000 ms)
            // for the callDuration parameter of TestMethod; a dummy variable
            // for the out parameter (threadId); the callback delegate; and
            // state information that can be retrieved by the callback method.
            // In this case, the state information is a string that can be used
            // to format a console message.
            IAsyncResult result = caller.BeginInvoke(3000,
                out dummy,
                new AsyncCallback(CallbackMethod),
                "The call executed on thread {0}, with return value \"{1}\".");

            Console.WriteLine("The main thread {0} continues to execute...",
                Thread.CurrentThread.ManagedThreadId);

            // The callback is made on a ThreadPool thread. ThreadPool threads
            // are background threads, which do not keep the application running
            // if the main thread ends. Comment out the next line to demonstrate
            // this.
            Thread.Sleep(4000);

            Console.WriteLine("The main thread ends.");
        }

        // The callback method must have the same signature as the
        // AsyncCallback delegate.
        static void CallbackMethod(IAsyncResult ar)
```

```
        {
            // Retrieve the delegate.
            AsyncResult result = (AsyncResult) ar;
            AsyncMethodCaller caller = (AsyncMethodCaller) result.AsyncDelegate;

            // Retrieve the format string that was passed as state
            // information.
            string formatString = (string) ar.AsyncState;

            // Define a variable to receive the value of the out parameter.
            // If the parameter were ref rather than out then it would have to
            // be a class-level field so it could also be passed to BeginInvoke.
            int threadId = 0;

            // Call EndInvoke to retrieve the results.
            string returnValue = caller.EndInvoke(out threadId, ar);

            // Use the format string to format the output message.
            Console.WriteLine(formatString, threadId, returnValue);
        }
    }
}

/* This example produces output similar to the following:

The main thread 1 continues to execute...
Test method begins.
The call executed on thread 3, with return value "My call time was 3000.".
The main thread ends.
 */
```

```
Imports System
Imports System.Threading
Imports System.Runtime.Remoting.Messaging

Namespace Examples.AdvancedProgramming.AsynchronousOperations

    Public Class AsyncMain

        Shared Sub Main()

            ' Create an instance of the test class.
            Dim ad As New AsyncDemo()

            ' Create the delegate.
            Dim caller As New AsyncMethodCaller(AddressOf ad.TestMethod)

            ' The threadId parameter of TestMethod is an <Out> parameter, so
            ' its input value is never used by TestMethod. Therefore, a dummy
            ' variable can be passed to the BeginInvoke call. If the threadId
            ' parameter were a ByRef parameter, it would have to be a class-
            ' level field so that it could be passed to both BeginInvoke and
            ' EndInvoke.
            Dim dummy As Integer = 0

            ' Initiate the asynchronous call, passing three seconds (3000 ms)
            ' for the callDuration parameter of TestMethod; a dummy variable
            ' for the <Out> parameter (threadId); the callback delegate; and
            ' state information that can be retrieved by the callback method.
            ' In this case, the state information is a string that can be used
            ' to format a console message.
            Dim result As IAsyncResult = caller.BeginInvoke(3000, _
                dummy, _
                AddressOf CallbackMethod, _
                "The call executed on thread {0}, with return value ""{1}""." )

            Console.WriteLine("The main thread {0} continues to execute...", _
```

```
                Thread.CurrentThread.ManagedThreadId)

            ' The callback is made on a ThreadPool thread. ThreadPool threads
            ' are background threads, which do not keep the application running
            ' if the main thread ends. Comment out the next line to demonstrate
            ' this.
            Thread.Sleep(4000)

            Console.WriteLine("The main thread ends.")
        End Sub

        ' The callback method must have the same signature as the
        ' AsyncCallback delegate.
        Shared Sub CallbackMethod(ByVal ar As IAsyncResult)
            ' Retrieve the delegate.
            Dim result As AsyncResult = CType(ar, AsyncResult)
            Dim caller As AsyncMethodCaller = CType(result.AsyncDelegate, AsyncMethodCaller)

            ' Retrieve the format string that was passed as state
            ' information.
            Dim formatString As String = CType(ar.AsyncState, String)

            ' Define a variable to receive the value of the <Out> parameter.
            ' If the parameter were ByRef rather than <Out> then it would have to
            ' be a class-level field so it could also be passed to BeginInvoke.
            Dim threadId As Integer = 0

            ' Call EndInvoke to retrieve the results.
            Dim returnValue As String = caller.EndInvoke(threadId, ar)

            ' Use the format string to format the output message.
            Console.WriteLine(formatString, threadId, returnValue)
        End Sub
    End Class
End Namespace

' This example produces output similar to the following:
'
'The main thread 1 continues to execute...
'Test method begins.
'The call executed on thread 3, with return value "My call time was 3000.".
'The main thread ends.
```

See Also

Delegate

Event-based Asynchronous Pattern (EAP)