Event-based Asynchronous Pattern Overview

.NET Framework (current version)

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 Handling and Raising 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:

VB

```
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
```

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

.NET Framework (current version)

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:

VB

```
Public Event MethodNameCompleted As MethodNameCompletedEventHandler
```

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

VB

```
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
```

   - 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:

VB

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

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

VB

```
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
```

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

Deciding When to Implement the Event-based Asynchronous Pattern

.NET Framework (current version)

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

.NET Framework (current version)

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.

VB

```
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.

VB

```
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.

VB

```
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.

VB

```
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.

   VB

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

VB

```
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.

VB

```
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.

VB

```
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`.

VB

```
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.

VB

```
' 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.

VB

```
' 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.

VB

```
' 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.

VB

```
' 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**.

   VB

```
    ' 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.

VB

```
' 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`.

VB

```
    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.

VB

```
' 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.

VB

```
' 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 Component That Supports the Event-based Asynchronous Pattern

.NET Framework (current version)

The following code example implements a component with an asynchronous method, according to the Event-based Asynchronous Pattern Overview. The component is a prime number calculator that uses the Sieve of Eratosthenes algorithm to determine if a number is prime or composite.

There is extensive support for this task in Visual Studio. For more information, see Walkthrough: Implementing a Component That Supports the Event-based Asynchronous Pattern.

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

Example

VB

```
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
```

VB

```


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.
    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
    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

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

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

.NET Framework (current version)

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

VB

```
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
        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.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.
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
```

```
    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

.NET Framework (current version)

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 (Windows Forms).

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.

VB

```
Friend WithEvents PictureBox1 As System.Windows.Forms.PictureBox
```

VB

```
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.

VB

```
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
```

VB

```
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