**Asynchronous Programming**

An asynchronous operation is an operation that runs on a separate thread; the thread that initiates an asynchronous operation does not need to wait for that operation to complete before it can continue.

Asynchronous operations are closely related to tasks. The .NET Framework 4.5 includes some new features that make it easier to perform asynchronous operations. These operations transparently create new tasks and coordinate their actions, enabling you to concentrate on the business logic of your application. In particular, the async and await keywords enable you to invoke an asynchronous operation and wait for the result within a single method, without blocking the thread.

In this module, you will learn various techniques for invoking and managing asynchronous operations.

**Using the Dispatcher**  
  
In the .NET Framework, each thread is associated with a Dispatcher object. The dispatcher is responsible for maintaining a queue of work items for the thread. When you work across multiple threads, for example, by running asynchronous tasks, you can use the Dispatcher object to invoke logic on a specific thread. You typically need to do this when you use asynchronous operations in graphical applications. For example, if a user clicks a button in a Windows® Presentation Foundation (WPF) application, the click event handler runs on the UI thread. If the event handler starts an asynchronous task, that task runs on the background thread. As a result, the task logic no longer has access to controls on the UI, because these are all owned by the UI thread. To update the UI, the task logic must use the Dispatcher.BeginInvoke method to queue the update logic on the UI thread.

Consider a simple WPF application that consists of a button named btnGetTime and a label named lblTime. When the user clicks the button, you use a task to get the current time. In this example, the task simply queries the DateTime.Now property, but in many scenarios, your applications might retrieve data from web services or databases in response to activity on the UI.

The following code example shows how you might attempt to update the UI from your task logic.

// The Wrong Way to Update a UI Object  
public void btnGetTime\_Click(object sender, RoutedEventArgs e)  
{  
   Task.Run(() =>   
      {  
         string currentTime = DateTime.Now.ToLongTimeString();  
         SetTime(currentTime);  
      }  
}  
private void SetTime(string time)  
{  
   lblTime.Content = time;  
}

If you were to run the preceding code, you would get an InvalidOperationException exception with the message ”The calling thread cannot access this object because a different thread owns it.” This is because the SetTime method is running on a background thread, but the lblTime label was created by the UI thread. To update the contents of the lblTime label, you must run the SetTime method on the UI thread.

To do this, you can retrieve the Dispatcher object that is associated with the lblTime object and then call the Dispatcher.BeginInvoke method to invoke the SetTime method on the UI thread.

The following code example shows how to use the Dispatcher.BeginInvoke method to update a control on the UI thread.

// The Correct Way to Update a UI Object  
public void buttonGetTime\_Click(object sender, RoutedEventArgs e)  
{  
   Task.Run(() =>   
      {  
         string currentTime = DateTime.Now.ToLongTimeString();  
         lblTime.Dispatcher.BeginInvoke(new Action(() => SetTime(currentTime)));  
      }  
}  
private void SetTime(string time)  
{  
   lblTime.Content = time;  
}

Note that the BeginInvoke method will not accept an anonymous delegate. The previous example uses the Action delegate to invoke the SetTime method. However, you can use any delegate that matches the signature of the method you want to call.

**Using async and await**  
  
The async and await keywords were introduced in the .NET Framework 4.5 to make it easier to perform asynchronous operations. You use the async modifier to indicate that a method is asynchronous. Within the async method, you use the await operator to indicate points at which the execution of the method can be suspended while you wait for a long-running operation to return. While the method is suspended at an await point, the thread that invoked the method can do other work.

Unlike other asynchronous programming techniques, the async and await keywords enable you to run logic asynchronously on a single thread. This is particularly useful when you want to run logic on the UI thread, because it enables you to run logic asynchronously on the same thread without blocking the UI.

Consider a simple WPF application that consists of a button named btnLongOperation and a label named lblResult. When the user clicks the button, the event handler invokes a long-running operation. In this example, it simply sleeps for 10 seconds and then returns the result “Operation complete.” In practice, however, you might make a call to a web service or retrieve information from a database. When the task is complete, the event handler updates the lblResult label with the result of the operation.

The following code example shows an application that performs a synchronous long-running operation on the UI thread.

// Running a Synchronous Operation on the UI Thread  
private void btnLongOperation\_Click(object sender, RoutedEventArgs e)  
{  
   lblResult.Content = "Commencing long-running operation…";  
   Task<string> task1 = Task.Run<string>(() =>  
      {  
         // This represents a long-running operation.  
         Thread.Sleep(10000);  
         return "Operation Complete";  
      }  
   // This statement blocks the UI thread until the task result is available.  
   lblResult.Content = task1.Result;  
}

In this example, the final statement in the event handler blocks the UI thread until the result of the task is available. This means that the UI will effectively freeze, and the user will be unable to resize the window, minimize the window, and so on. To enable the UI to remain responsive, you can convert the event handler into an asynchronous method.

The following code example shows an application that performs an asynchronous long-running operation on the UI thread.

// Running an Asynchronous Operation on the UI Thread  
private async void btnLongOperation\_Click(object sender, RoutedEventArgs e)  
{  
   lblResult.Content = "Commencing long-running operation…";  
   Task<string> task1 = Task.Run<string>(() =>  
      {  
         // This represents a long-running operation.  
         Thread.Sleep(10000);  
         return "Operation Complete";  
      }  
   // This statement is invoked when the result of task1 is available.  
   // In the meantime, the method completes and the thread is free to do other work.  
   lblResult.Content = await task1;  
}

This example includes two key changes from the previous example:

* The method declaration now includes the async keyword.
* The blocking statement has been replaced by an await operator.

Notice that when you use the await operator, you do not await the result of the task—you await the task itself. When the .NET runtime executes an async method, it effectively bypasses the await statement until the result of the task is available. The method returns and the thread is free to do other work. When the result of the task becomes available, the runtime returns to the method and executes the await statement.

**Creating Awaitable Methods**  
  
The await operator is always used to await the completion of a Task instance in a non-blocking manner. If you want to create an asynchronous method that you can wait for with the await operator, your method must return a Task object. When you convert a synchronous method to an asynchronous method, use the following guidelines:

* If your synchronous method returns void (in other words, it does not return a value), the asynchronous method should return a Task object.
* If your synchronous method has a return type of TResult, your asynchronous method should return a Task<TResult> object.

An asynchronous method can return void; however, this is typically only used for event handlers. Wherever possible, you should return a Task object to enable callers to use the await operator with your method.

The following code example shows a synchronous method that waits ten seconds and then returns a string.

// A Long-Running Synchronous Method  
private string GetData()  
{  
   var task1 = Task.Run<string>(() =>  
      {  
         // Simulate a long-running task.  
         Thread.Sleep(10000);  
         return "Operation Complete.";  
      });  
   return task1.Result;  
}

To convert this into an awaitable asynchronous method, you must:

* Add the async modifier to the method declaration.
* Change the return type from string to Task<string>.
* Modify the method logic to use the await operator with any long-running operations.

The following code example shows how to convert the previous synchronous method into an asynchronous method.

// Creating an Awaitable Asynchronous Method  
private async Task<string> GetData()  
{  
   var result = await Task.Run<string>(() =>  
      {  
         // Simulate a long-running task.  
         Thread.Sleep(10000);  
         return "Operation Complete.";  
      });  
   return result;  
}

The GetData method returns a task, so you can use the method with the await operator. For example, you might call the method in the event handler for the click event of a button and use the result to set the value of a label named lblResult.

The following code example shows how to call an awaitable asynchronous method.

// Calling an Awaitable Asynchronous Method  
private async void btnGetData\_Click(object sender, RoutedEventArgs e)  
{  
   lblResult.Content = await GetData();  
}

Note that you can only use the await keyword in an asynchronous method.  
  
**Creating and Invoking Callback Methods**  
  
If you must run complex logic when an asynchronous method completes, you can configure your asynchronous method to invoke a callback method. The asynchronous method passes data to the callback method, and the callback method processes the data. You might also use the callback method to update the UI with the processed results.

To configure an asynchronous method to invoke a callback method, you must include a delegate for the callback method as a parameter to the asynchronous method. A callback method typically accepts one or more arguments and does not return a value. This makes the Action<T> delegate a good choice to represent a callback method, where T is the type of your argument. If your callback method requires multiple arguments, there are versions of the Action delegate that accept up to 16 type parameters.

Consider a WPF application that consists of a button named btnGetCoffees and a list named lstCoffees. When the user clicks the button, the event handler invokes an asynchronous method that retrieves a list of coffees. When the asynchronous data retrieval is complete, the method invokes a callback method. The callback method removes any duplicates from the results and then displays the updated results in the listCoffees list.

The following code example shows an asynchronous method that invokes a callback method.

// Invoking a Callback Method  
// This is the click event handler for the button.  
private async void btnGetCoffees\_Click(object sender, RoutedEventArgs e)  
{  
   await GetCoffees(RemoveDuplicates);  
}  
// This is the asynchronous method.  
public async Task GetCoffees(Action<IEnumerable<string>> callback)  
{  
   // Simulate a call to a database or a web service.  
   var coffees = await Task.Run(() =>  
      {  
         var coffeeList = new List<string>();  
         coffeeList.Add("Caffe Americano");  
         coffeeList.Add("Café au Lait");  
         coffeeList.Add("Café au Lait");  
         coffeeList.Add("Espresso Romano");  
         coffeeList.Add("Latte");  
         coffeeList.Add("Macchiato");  
         return coffeeList;  
      }  
   // Invoke the callback method asynchronously.  
   await Task.Run(() => callback(coffees));  
}  
// This is the callback method.  
private void RemoveDuplicates(IEnumerable<string> coffees)  
{  
   IEnumerable<string> uniqueCoffees = coffees.Distinct();   
   Dispatcher.BeginInvoke(new Action(() =>  
      {  
         lstCoffees.ItemsSource = uniqueCoffees;  
      }  
}

In the previous example, the RemoveDuplicates callback method accepts a single argument of type IEnumerable<string> and does not return a value. To support this callback method, you add a parameter of type Action<IEnumerable<string>> to your asynchronous method. When you invoke the asynchronous method, you supply the name of the callback method as an argument.

**Working with APM Operations**  
  
Many .NET Framework classes that support asynchronous operations do so by implementing a design pattern known as APM. The APM pattern is typically implemented as two methods: a BeginOperationName method that starts the asynchronous operation and an EndOperationName method that provides the results of the asynchronous operation. You typically call the EndOperationName method from within a callback method. For example, the HttpWebRequest class includes methods named BeginGetResponse and EndGetResponse. The BeginGetResponse method submits an asynchronous request to an Internet or intranet resource, and the EndGetResponse method returns the actual response that the Internet resource provides.

Classes that implement the APM pattern use an IAsyncResult instance to represent the status of the asynchronous operation. The BeginOperationName method returns an IAsyncResult object, and the EndOperationName method includes an IAsyncResult parameter.  
The Task Parallel Library makes it easier to work with classes that implement the APM pattern. Rather than implementing a callback method to call the EndOperationName method, you can use the TaskFactory.FromAsync method to invoke the operation asynchronously and return the result in a single statement. The TaskFactory.FromAsync method includes several overloads to accommodate APM methods that take varying numbers of arguments.

Consider a WPF application that verifies URLs that a user provides. The application consists of a box named txtUrl, a button named btnSubmitUrl, and a label named lblResults. The user types a URL in the box and then clicks the button. The click event handler for the button submits an asynchronous web request to the URL and then displays the status code of the response in the label. Rather than implementing a callback method to handle the response, you can use the TaskFactory.FromAsync method to perform the entire operation.

The following code example shows how to use the TaskFactory.FromAsync method to submit an asynchronous web request and handle the response.

// Using the TaskFactory.FromAsync Method  
private async void btnCheckUrl\_Click(object sender, RoutedEventArgs e)  
{  
   // Get the URL provided by the user.  
   string url = txtUrl.Text;  
   // Create an HTTP request.  
   HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);  
   // Submit the request and await a response.  
   HttpWebResponse response =   
       await Task<WebResponse>.Factory.FromAsync(request.BeginGetResponse, request.EndGetResponse, request)   
          as HttpWebResponse;  
   // Display the status code of the response.  
   lblResult.Content = String.Format("The URL returned the following status code: {0}", response.StatusCode);  
}

**Handling Exceptions from Awaitable Methods**  
  
When you perform asynchronous operations with the async and await keywords, you can handle exceptions in the same way that you handle exceptions in synchronous code, which is by using try/catch blocks.

The following code example shows how to catch an exception that an awaitable method has thrown.

// Catching an Awaitable Method Exception  
private async void btnThrowError\_Click(object sender, RoutedEventArgs e)  
{  
   using (WebClient client = new WebClient())  
   {  
      try  
      {  
         string data = await client.DownloadStringTaskAsync("http://fourthcoffee/bogus");  
      }  
      catch (WebException ex)  
      {  
         lblResult.Content = ex.Message;  
      }  
   }  
}

In the previous example, the click event handler for a button calls the WebClient.DownloadStringTaskAsync method asynchronously by using the await operator. The URL that is provided is invalid, so the method throws a WebException exception. Even though the operation is asynchronous, control returns to the btnThrowError\_Click method when the asynchronous operation is complete and the exception is handled correctly. This works because behind the scenes, the Task Parallel Library is catching the asynchronous exception and re-throwing it on the UI thread.

**Unobserved Exceptions**

When a task raises an exception, you can only handle the exception when the joining thread accesses the task, for example, by using the await operator or by calling the Task.Wait method. If the joining thread never accesses the task, the exception will remain unobserved. When the .NET Framework garbage collector (GC) detects that a task is no longer required, the task scheduler will throw an exception if any task exceptions remain unobserved. By default, this exception is ignored. However, you can implement an exception handler of last resort by subscribing to the TaskScheduler.UnobservedTaskException event. Within the exception handler, you can set the status of the exception to Observed to prevent any further propagation.

The following code example shows how to subscribe to the TaskScheduler.UnobservedTaskException event.

// Implementing a Last-Resort Exception Handler  
static void Main(string[] args)  
{  
   // Subscribe to the TaskScheduler.UnobservedTaskException event and define an event handler.  
   TaskScheduler.UnobservedTaskException += (object sender, UnobservedTaskExceptionEventArgs e) =>  
      {  
         foreach (Exception ex in ((AggregateException)e.Exception).InnerExceptions)  
         {  
            Console.WriteLine(String.Format("An exception occurred: {0}", ex.Message));  
         }  
         // Set the exception status to Observed.  
         e.SetObserved();  
      }  
   // Launch a task that will throw an unobserved exception   
   // by attempting to download an item from an invalid URL.  
   Task.Run(() =>  
      {  
         using(WebClient client = new WebClient())  
         {  
            client.DownloadStringTaskAsync("http://fourthcoffee/bogus");  
         }  
      });  
   // Give the task time to complete and then trigger garbage collection (for example purposes only).  
   Thread.Sleep(5000);  
   GC.WaitForPendingFinalizers();  
   GC.Collect();  
   Console.WriteLine("Execution complete.");  
   Console.ReadLine();  
}

If you use a debugger to step through this code, you will see that the UnobservedTaskException event is fired when the GC runs.  
In the .NET Framework 4.5, the .NET runtime ignores unobserved task exceptions by default and allows your application to continue executing. This contrasts with the default behavior in the .NET Framework 4.0, where the .NET runtime would terminate any processes that throw unobserved task exceptions. You can revert to the process termination approach by adding a ThrowUnobservedTaskExceptions element to your application configuration file.

The following code example shows how to add a ThrowUnobservedTaskExceptions element to an application configuration file.

// Configuring the ThrowUnobservedTaskExceptions Element  
<configuration>  
   …  
   <runtime>  
      <ThrowUnobservedTaskExceptions enabled="true" />  
   </runtime>  
</configuration>

If you set ThrowUnobservedTaskExceptions to true, the .NET runtime will terminate any processes that contain unobserved task exceptions. A recommended best practice is to set this flag to true during application development and to remove the flag before you release your code.