Module 10

Improving Application Performance and Responsiveness

Module Overview

- Implementing Multitasking
- Performing Operations Asynchronously
- Synchronizing Concurrent Access to Data

Lesson 1: Implementing Multitasking

- Creating Tasks
- Controlling Task Execution
- Returning a Value from a Task
- Cancelling Long-Running Tasks
- Running Tasks in Parallel
- Linking Tasks
- Handling Task Exceptions

Creating Tasks

- Use an **Action** delegate

```
Task task1 = new Task(new Action(MyMethod));
```

- Use an anonymous delegate/anonymous method

```
Task task2 = new Task(delegate
{
   Console.WriteLine("Task 2 reporting");
});
```

- Use lambda expressions (recommended)

```
Task task2 = new Task(() =>
{
   Console.WriteLine(" Task 2 reporting");
});
```

Controlling Task Execution

- To start a task:
  - **Task.Start** instance method
  - **Task.Factory.StartNew** static method
  - **Task.Run** static method
- To wait for tasks to complete:
  - **Task.Wait** instance method
  - **Task.WaitAll** static method
  - **Task.WaitAny** static method

Returning a Value from a Task

- Use the **Task<TResult>** class
- Specify the return type in the type argument

```
Task<string> task1 = Task.Run<string>( () =>
  DateTime.Now.DayOfWeek.ToString() );
```

- Get the result from the **Result** property

```
Console.WriteLine("Today is {0}", task1.Result);
```

Cancelling Long-Running Tasks

- Pass a cancellation token as an argument to the delegate method
- Request cancellation from the joining thread
- In the delegate method, check whether the cancellation token is cancelled
- Return or throw an **OperationCanceledException** exception

Running Tasks in Parallel

- Use **Parallel.Invoke** to run multiple tasks simultaneously

```
Parallel.Invoke( () => MethodForFirstTask(),
                 () => MethodForSecondTask(),
                 () => MethodForThirdTask() );
```

- Use **Parallel.For** to run **for** loop iterations in parallel
- Use **Parallel.ForEach** to run **foreach** loop iterations in parallel
- Use PLINQ to run LINQ expressions in parallel

Linking Tasks

- Use task continuations to chain tasks together:
  - **Task.ContinueWith** method links continuation task to antecedent task
  - Continuation task starts when antecedent task completes
  - Antecedent task can pass result to continuation task
- Use nested tasks if you want to start an *independent* task from a task delegate
- Use child tasks if you want to start a *dependent* task from a task delegate

Handling Task Exceptions

- Call **Task.Wait** to catch propagated exceptions
- Catch **AggregateException** in the **catch** block
- Iterate the **InnerExceptions** property and handle individual exceptions

```
try
{
   task1.Wait();
}
catch(AggregateException ae)
{
   foreach(var inner in ae.InnerExceptions)
   {
      // Deal with each exception in turn.
   }
}
```

Lesson 2: Performing Operations Asynchronously

- Using the Dispatcher
- Using async and await
- Creating Awaitable Methods
- Creating and Invoking Callback Methods
- Working with APM Operations
- Demonstration: Using the Task Parallel Library to Invoke APM Operations
- Handling Exceptions from Awaitable Methods

Using the Dispatcher

- To update a UI element from a background thread:
  - Get the **Dispatcher** object for the thread that owns the UI element
  - Call the **BeginInvoke** method
  - Provide an **Action** delegate as an argument

```
lblTime.Dispatcher.BeginInvoke(new Action(() =>
   SetTime(currentTime)));
```

Using async and await

- Add the **async** modifier to method declarations
- Use the **await** operator within **async** methods to wait for a task to complete without blocking the thread

```
private async void btnLongOperation_Click(object sender,
RoutedEventArgs e)
{
   ...
   Task<string> task1 = Task.Run<string>(() =>
     {
        ...
     }
  lblResult.Content = await task1;
}
```

Creating Awaitable Methods

- The **await** operator is always used to wait for a task to complete
- If your synchronous method returns **void**, the asynchronous equivalent should return **Task**
- If your synchronous method has a return type of **T**, the asynchronous equivalent should return **Task<T>**

Creating and Invoking Callback Methods

- Use the **Action<*T*>** delegate to represent your callback method
- Add the delegate to your asynchronous method parameters

```
public async Task
GetCoffees(Action<IEnumerable<string>> callback)
```

- Invoke the delegate asynchronously within your method

```
await Task.Run(() => callback(coffees));
```

Working with APM Operations

- Use the **TaskFactory.FromAsync** method to call methods that implement the APM pattern

```
HttpWebRequest request =
   (HttpWebRequest)WebRequest.Create(url);

HttpWebResponse response =
   await Task<WebResponse>.Factory.FromAsync(
      request.BeginGetResponse,
      request.EndGetResponse,
      request) as HttpWebResponse;
```

Demonstration: Using the Task Parallel Library to Invoke APM Operations

- In this demonstration, you will learn how to:
  - Use a conventional approach to invoke APM operations
  - Use the Task Parallel Library to invoke APM operations
  - Compare the two approaches

Handling Exceptions from Awaitable Methods

- Use a conventional **try/catch** block to catch exceptions in asynchronous methods
- Subscribe to the **TaskScheduler.UnobservedTaskException** event to create an event handler of last resort

```
TaskScheduler.UnobservedTaskException +=
  (object sender, UnobservedTaskExceptionEventArgs e) =>
    {
      // Respond to the unobserved task exception.
    }
```

Lesson 3: Synchronizing Concurrent Access to Data

- Using Locks
- Demonstration: Using Lock Statements
- Using Synchronization Primitives with the Task Parallel Library
- Using Concurrent Collections
- Demonstration: Improving the Responsiveness and Performance of the Application Lab

Using Locks

- Create a private object to apply the lock to
- Use the **lock** statement and specify the locking object
- Enclose your critical section of code in the **lock** block

```
private object lockingObject = new object();
lock (lockingObject)
{
   // Only one thread can enter this block at any one time.
}
```

Demonstration: Using Lock Statements

- In this demonstration, you will see how to:
  - Use **lock** statements to apply mutual-exclusion locks to critical sections of code
  - Observe the consequences if you omit the **lock** statement

Using Synchronization Primitives with the Task Parallel Library

- Use the **ManualResetEventSlim** class to limit resource access to one thread at a time
- Use the **SemaphoreSlim** class to limit resource access to a fixed number of threads
- Use the **CountdownEvent** class to block a thread until a fixed number of tasks signal completion
- Use the **ReaderWriterLockSlim** class to allow multiple threads to read a resource or a single thread to write to a resource at any one time
- Use the **Barrier** class to block multiple threads until they all satisfy a condition

Using Concurrent Collections

The **System.Collections.Concurrent** namespace includes generic classes and interfaces for thread-safe collections:

- **ConcurrentBag<T>**
- **ConcurrentDictionary<TKey, TValue>**
- **ConcurrentQueue<T>**
- **ConcurrentStack<T>**
- **IProducerConsumerCollection<T>**
- **BlockingCollection<T>**

Demonstration: Improving the Responsiveness and Performance of the Application Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Improving the Responsiveness and Performance of the Application

- Exercise 1: Ensuring That the UI Remains Responsive When Retrieving Teacher Data
- Exercise 2: Providing Visual Feedback During Long-Running Operations

Logon Information

- Virtual Machine: 20483B-SEA-DEV11, MSL-TMG1
- User Name: Student
- Password: Pa$$w0rd

Estimated Time: 75 minutes

Lab Scenario

You have been asked to update the Grades application to ensure that the UI remains responsive while the user is waiting for operations to complete. To achieve this improvement in responsiveness, you decide to convert the logic that retrieves the list of students for a teacher to use asynchronous methods. You also decide to provide visual feedback to the user to indicate when an operation is taking place.

Module Review and Takeaways

- Review Question(s)