Module 10: Improving Application Performance and Responsiveness

Contents:

Module overview

Modern processors use threads to concurrently run multiple operations. If your application performs all of its logic on a single thread, you do not make the best use of the available processing resources, which can result in a poor experience for your users. In this module, you will learn how to improve the performance of your applications by distributing your operations across multiple threads.

Objectives

After completing this module, you will be able to:

- Use the Task Parallel Library to implement multitasking.
- Perform long-running operations without blocking threads.
- Control how multiple threads can access resources concurrently.

Lesson 1: Implementing Multitasking

A typical graphical application consists of blocks of code that run when an event occurs; these events fire in response to actions such as the user clicking a button, moving the mouse, or opening a window. By default, this code runs by using the UI thread. However, you should avoid executing long-running operations on this thread because they can cause the UI to become unresponsive. Also, running all of your code on a single thread does not make good use of available processing power in the computer; most modern machines contain multiple processor cores, and running all operations on a single thread will only use a single processor core.

The Microsoft® .NET Framework now includes the Task Parallel Library. This is a set of classes that makes it easy to distribute your code execution across multiple threads. You can run these threads on different processor cores and take advantage of the parallelism that this model provides. You can assign long-running tasks to a separate thread, leaving the UI thread free to respond to user actions.

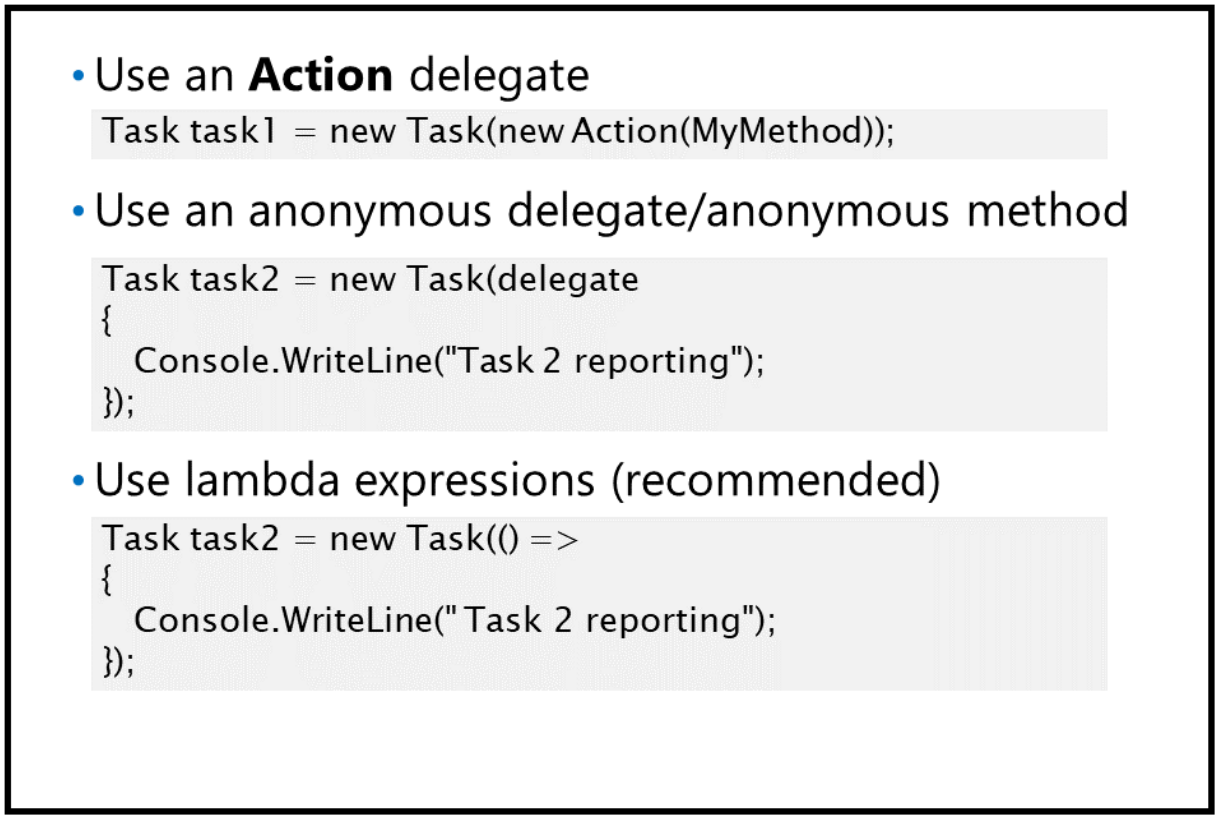
In this lesson, you will learn how to use the Task Parallel Library to create multithreaded and responsive applications.

Lesson objectives

After completing this lesson, you will be able to:

- Create tasks.
- Control how tasks are executed.
- Return values from tasks.
- Cancel long-running tasks.
- Run multiple tasks in parallel.
- Link tasks together.
- Handle exceptions that tasks throw.

Creating Tasks



The **Task** class lies at the heart of the Task Parallel Library in the .NET Framework. As the name suggests, you use the **Task** class to represent a task, or in other words, a unit of work. The **Task** class enables you to perform multiple tasks concurrently, each on a different thread. Behind the scenes, the Task Parallel Library manages the thread pool and assigns tasks to threads. You can implement sophisticated multitasking functionality by using the Task Parallel Library to chain tasks, pause tasks, wait for tasks to complete before continuing, and perform many other operations.

Creating a Task

You create a new **Task** object by using the **Task** class. A **Task** object runs a block of code, and you specify this code as a parameter to the constructor. You can provide this code in a method and create an **Action** delegate that wraps this method.

> **Note:** The **.NET Framework class library** contains the built-in **Action** delegate that lets you quickly define a delegate that has no return value and up to 16 parameters that use generic parameter types. It also provides the **Func** delegate that returns a result. You can use these delegates instead of defining new delegates for specific cases.

The following code example shows how to create a task by using an **Action** delegate:

Creating a Task by Using an Action Delegate

```
Task task1 = new Task(new Action(GetTheTime));
private static void GetTheTime()
{
   Console.WriteLine("The time now is {0}", DateTime.Now);
}
```

Using an **Action** delegate requires that you have defined a method that contains the code that you want to run in a task. However, if the sole purpose of this method is to provide the logic for a task and it is not reused anywhere else, you can find yourself creating (and having to remember the names of) a substantial number of methods. This makes maintenance more difficult. A more common approach is to use an anonymous method. An anonymous method is a method without a name, and you provide the code for an anonymous method inline, at the point you need to use it. You can use the **delegate** keyword to convert an anonymous method into a delegate.

The following code example shows how to create a task by using an anonymous delegate.

Creating a Task by Using an Anonymous Delegate

```
Task task2 = new Task( delegate { Console.WriteLine("The time now is
{0}", DateTime.Now); });
```

Using Lambda Expressions to Create Tasks

A lambda expression is a shorthand syntax that provides a simple and concise way to define anonymous delegates. When you create a **Task** instance, you can use a lambda expression to define the delegate that you want to associate with your task.

If you want your delegate to invoke a named method or a single line of code, you use can use a lambda expression. A lambda expression provides a shorthand notation for defining a delegate that can take parameters and return a result. It has the following form:

(input parameters) => expression

In this case:

- The lambda operator, **=>**, is read as “goes to.”
- The left side of the lambda operator includes any variables that you want to pass to the expression. If you do not require any inputs—for example, if you are invoking a method that takes no parameters—you include empty parentheses () on the left side of the lambda operator.

The right side of the lambda operator includes the expression you want to evaluate. This could be a comparison of the input parameters—for example, the expression **(x, y) => x == y** will return **true** if **x** is equal to **y**; otherwise, it will return **false**. Alternatively, you can call a method on the right side of the lambda operator.

The following code example shows how to use lambda expressions to represent a delegate that invokes a named method.

Using a Lambda Expression to Invoke a Named Method

```
Task task1 = new Task ( () => MyMethod() );
// This is equivalent to: Task task1 = new Task( delegate(MyMethod)
);
```

A lambda expression can be a simple expression or function call, as the previous example shows, or it can reference a more substantial block of code. To do this, specify the code in curly braces (like the body of a method) on the right side of the lambda operator:

(input parameters) => { Visual C# statements; }

The following code example shows how to use lambda expressions to represent a delegate that invokes an anonymous method.

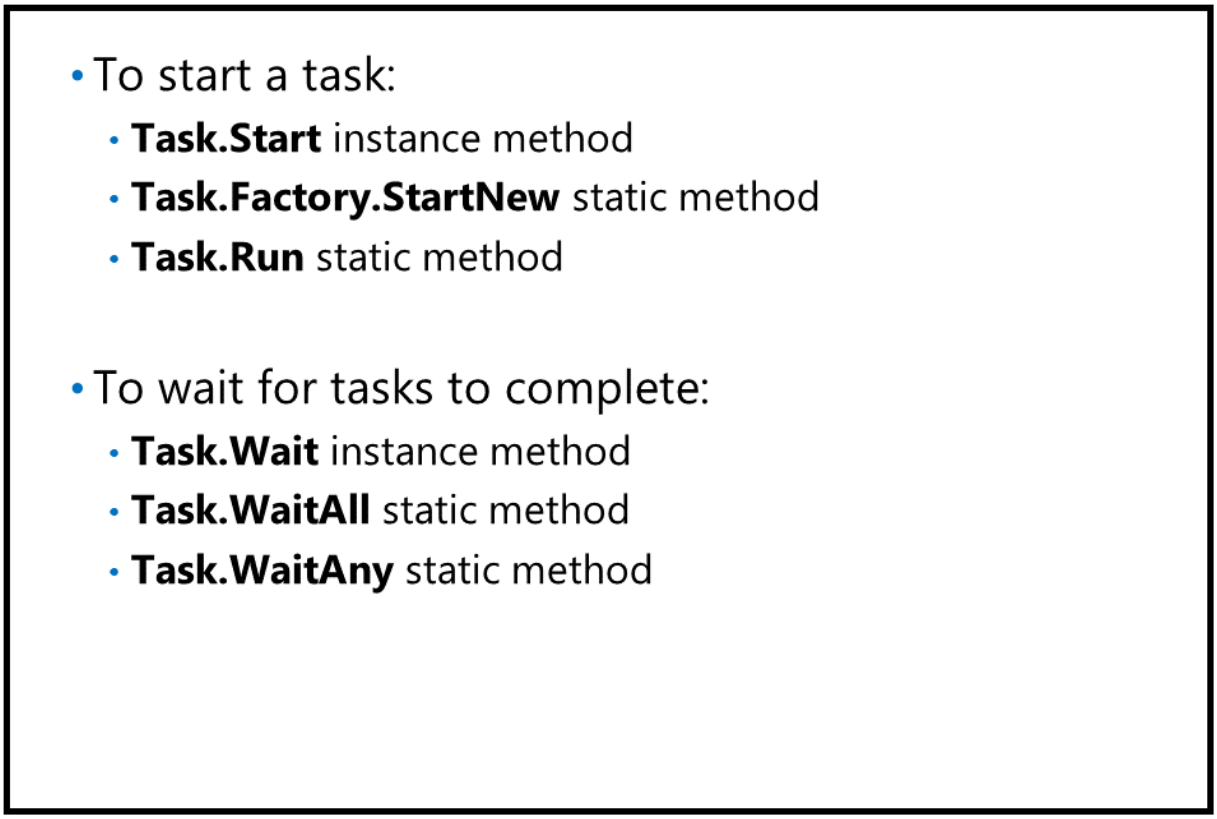
Using a Lambda Expression to Invoke an Anonymous Method

```
Task task2 = new Task( () => { Console.WriteLine("Test") } );
// This is equivalent to: Task task2 = new Task( delegate {
Console.WriteLine("Test") } );
```

As your delegates become more complex, lambda expressions offer a far more concise and easily understood way to express anonymous delegates and anonymous methods. As such, lambda expressions are the recommended approach when you work with tasks.

> **Reference Links:** For more information about lambda expressions, refer Lambda Expressions at **https://aka.ms/moc-20483c-m10-pg1**.

Controlling Task Execution



The Task Parallel Library offers several different approaches that you can use to start tasks. There are also various different ways in which you can pause the execution of your code until one or more tasks have completed.

Starting Tasks

When your code starts a task, the Task Parallel Library assigns a thread to your task and starts running that task. The task runs on a separate thread, so your code does not need to wait for the task to complete. Instead, the task and the code that invoked the task continue to run in parallel.

If you want to queue the task immediately, you use the **Start** method.

Using the Task.Start Method to Queue a Task

```
var task1 = new Task( () => Console.WriteLine("Task 1 has
completed.") );
task1.Start();
```

Alternatively, you can use the static **TaskFactory** class to create and queue a task with a single line of code. The **TaskFactory** class is exposed through the static **Factory** property of the **Task** class.

Using the TaskFactory.StartNew Method to Queue a Task

```
var task3 = Task.Factory.StartNew( () => Console.WriteLine("Task 3
has completed.") );
```

The **Task.Factory.StartNew** method is highly configurable and accepts a wide range of parameters. If you simply want to queue some code with the default scheduling options, you can use the static **Task.Run** method as a shortcut for the **Task.Factory.StartNew** method.

Using the Task.Run Method to Queue a Task

```
var task4 = Task.Run( () => Console.WriteLine("Task 4 has completed.
") );
```

Waiting for Tasks

In some cases, you may need to pause the execution of your code until a particular task has completed. Typically you do this if your code depends on the result from one or more tasks, or if you need to handle exceptions that a task may throw. The **Task** class offers various mechanisms to do this:

- If you want to wait for a single task to finish executing, use the **Task.Wait** method.
- If you want to wait for multiple tasks to finish executing, use the static **Task.WaitAll** method.
- If you want to wait for any one of a collection of tasks to finish executing, use the static **Task.WaitAny** method.

The following code example shows how to wait for a single task to complete.

Waiting for a Single Task to Complete

```
var task1 = Task.Run( () => LongRunningMethod() );
// Do some other work.
// Wait for task 1 to complete.
task1.Wait();
// Continue with execution.
```

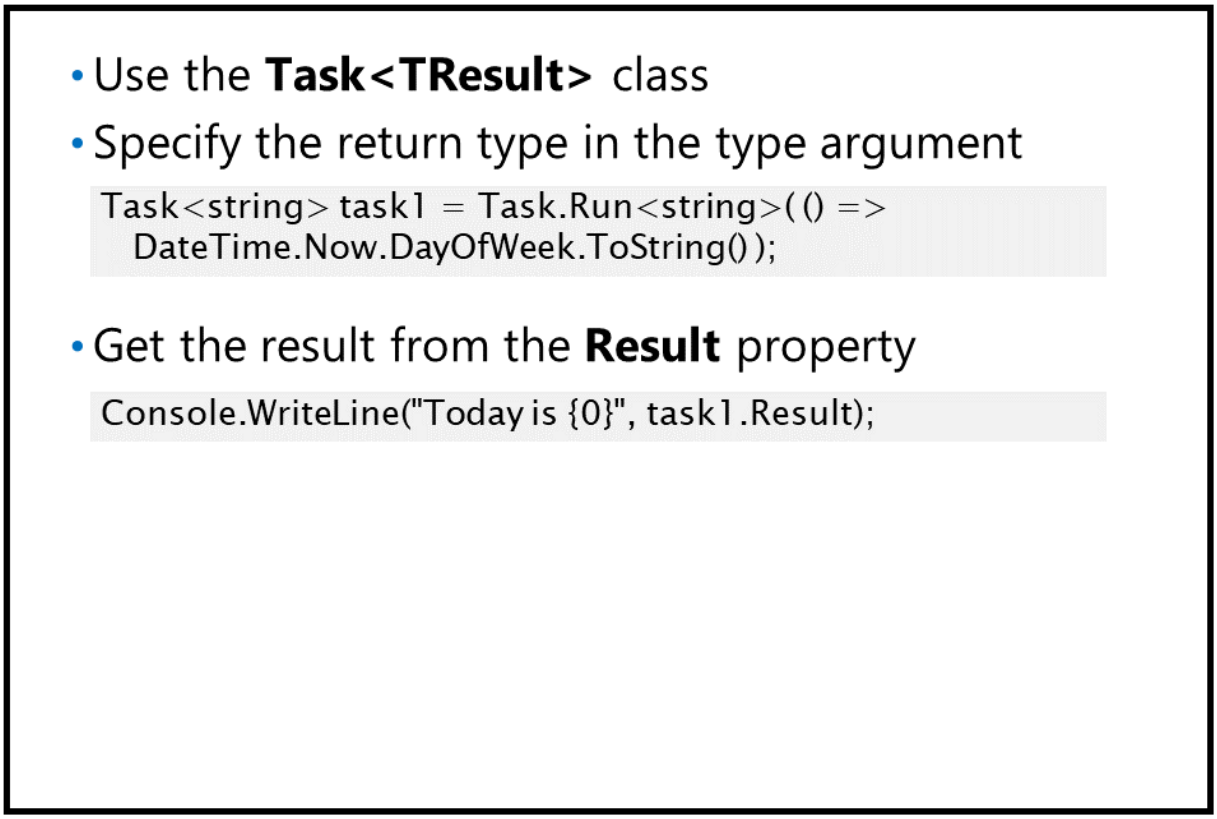

If you want to wait for multiple tasks to finish executing, or for one of a collection of tasks to finish executing, you must add your tasks to an array. You can then pass the array of tasks to the static **Task.WaitAll** or **Task.WaitAny** methods.

The following code example shows how to wait for multiple tasks to complete.

Waiting for Multiple Tasks to Complete

```
Task[] tasks = new Task[3]
{
   Task.Run( () => LongRunningMethodA()),
   Task.Run( () => LongRunningMethodB()),
   Task.Run( () => LongRunningMethodC())
};
// Wait for any of the tasks to complete.
Task.WaitAny(tasks);
// Alternatively, wait for all of the tasks to complete.
Task.WaitAll(tasks);
// Continue with execution.
```

Returning a Value from a Task



For tasks to be effective in real-world scenarios, you need to be able to create tasks that can return values, or *results*, to the code that initiated the task. The regular **Task** class does not enable you to do this. However, the Task Parallel Library also includes the generic **Task<TResult>** class that you can use when you need to return a value.

When you create an instance of **Task<TResult>**, you use the type parameter to specify the type of the result that the task will return. The **Task<TResult>** class exposes a read-only property named **Result**. After the task has finished executing, you can use the **Result** property to retrieve the return value of the task. The **Result** property is the same type as the task's type parameter.

The following example shows how to use the **Task<TResult>** class.

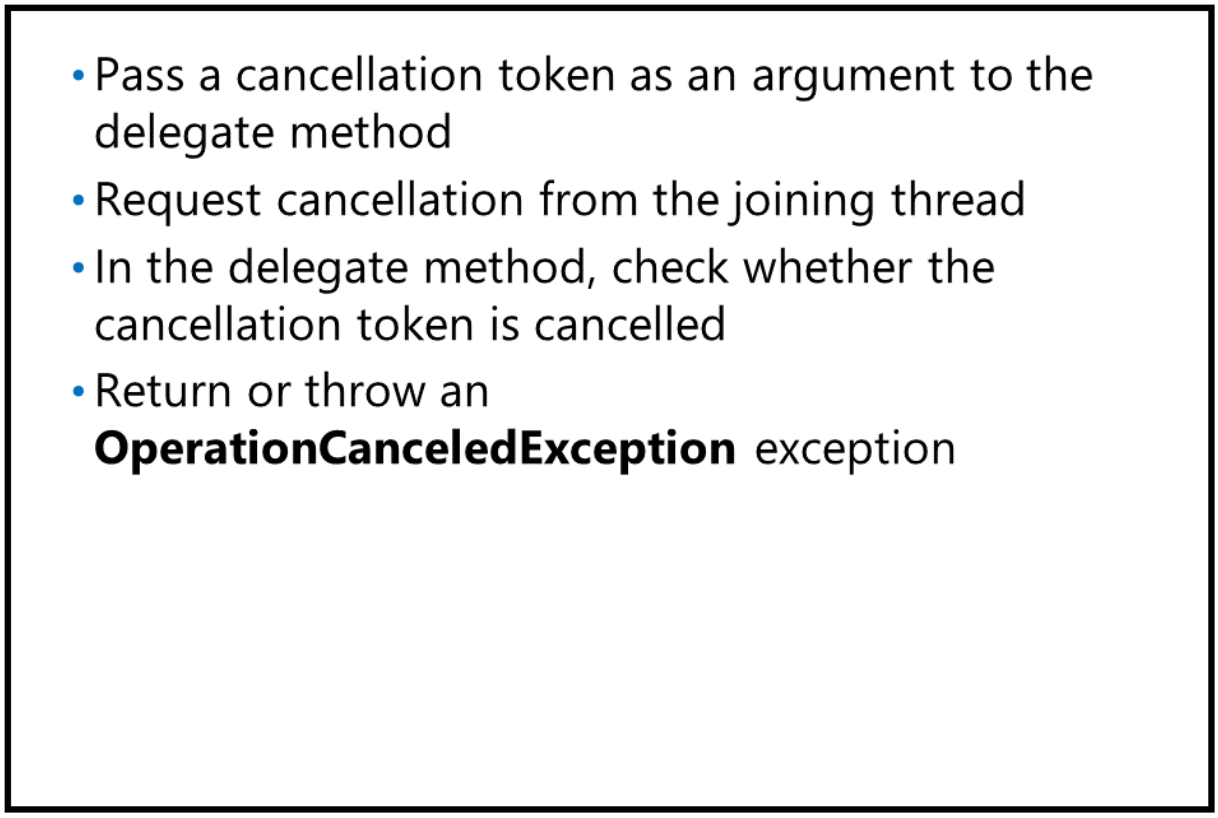
Retrieving a Value from a Task

```
// Create and queue a task that returns the day of the week as a
string.
Task<string> task1 = Task.Run<string>( () =>
DateTime.Now.DayOfWeek.ToString() );
```

```
// Retrieve and display the task result.
Console.WriteLine(task1.Result);
```

If you access the **Result** property before the task has finished running, your code will wait until a result is available before proceeding.

Cancelling Long-Running Tasks



Tasks are often used to perform long-running operations without blocking the UI thread, because of their asynchronous nature. In some cases, you will want to give your users the opportunity to cancel a task if they are tired of waiting. However, it would be dangerous to simply abort the task on demand, because this could leave your application data in an unknown state. Instead, the Task Parallel Library uses *cancellation tokens* to support a cooperative cancellation model. At a high level, the cancellation process works as follows:

1. When you create a task, you also create a cancellation token.
2. You pass the cancellation token as an argument to your delegate method.
3. On the thread that created the task, you *request* cancellation by calling the **Cancel** method on the cancellation token source.

4. In your task method, you can check the status of the cancellation token at any point. If the instigator has requested that the task be cancelled, you can terminate your task logic gracefully, possibly rolling back any changes resulting from the work that the task has performed.

Typically, you would check whether the cancellation token has been set to canceled at one or more convenient points in your task logic. For example, if your task logic iterates over a collection, you might check for cancellation after each iteration.

The following code example shows how to cancel a task.

Cancelling a Task

```
// Create a cancellation token source and obtain a cancellation
token.
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken ct = cts.Token;
// Create and start a task.
Task.Run( () => doWork(ct) );
// Method run by the task.
private void doWork(CancellationToken token);
{
   …
   // Check for cancellation.
   if(token.IsCancellationRequested)
   {
      // Tidy up and finish.
      …
      return;
   }
   // If the task has not been cancelled, continue running as
normal.
   …
}
```

This approach works well if you do not need to check whether the task ran to completion. Each task exposes a **Status** property that enables you to monitor the current status of the task in the task life cycle. If you cancel a task by returning the task method, as shown in the previous example, the task status is set to **RanToCompletion**. In other words, the task has no way of knowing why the method returned—it may have returned in response to a cancellation request, or it may simply have completed its logic.

If you want to cancel a task and be able to confirm that it was cancelled, you need to pass the cancellation token as an argument to the task constructor in addition to the delegate method. In your task method, you check the status of the cancellation token. If the instigator has requested the cancellation of the task, you throw an **OperationCanceledException** exception. When an **OperationCanceledException** exception occurs, the Task Parallel Library checks the cancellation token to verify whether a cancellation was requested. If it was, the Task Parallel Library handles the **OperationCanceledException** exception, sets the task status to **Canceled**, and throws a **TaskCanceledException** exception. In the code that created the cancellation request, you can catch this **TaskCanceledException** exception and deal with the cancellation accordingly.

To check whether a cancellation was requested and throw an **OperationCanceledException** exception if it was, you call the **ThrowIfCancellationRequested** method on the cancellation token.

The following code example shows how to cancel a task by throwing an **OperationCanceledException** exception.

Canceling a Task by Throwing an Exception

```
// Create a cancellation token source and obtain a cancellation
token.
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken ct = cts.Token;
// Create and start a task.
Task.Run( () => doWork(ct) );
```

```
// Method run by the task.
private void doWork(CancellationToken token);
{
   …
   // Throw an OperationCanceledException if cancellation was
requested.
   token.ThrowIfCancellationRequested();
   // If the task has not been cancelled, continue running as
normal.
   …
}
```

Reference Links: For more information about cancelling tasks, refer Task Cancellation at **http://go.microsoft.com/fwlink/?LinkID=267837** and How to: Cancel a Task and Its Children at **http://go.microsoft.com/fwlink/?LinkID=267838**.

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

The Task Parallel Library includes a static class named **Parallel**. The **Parallel** class provides a range of methods that you can use to execute tasks simultaneously.

Executing a Set of Tasks Simultaneously

If you want to run a fixed set of tasks in parallel, you can use the **Parallel.Invoke** method. When you call this method, you use lambda expressions to specify the tasks that you want to run simultaneously. You do not need to explicitly create each task—the tasks are created implicitly from the delegates that you supply to the **Parallel.Invoke** method.

The following code example shows how to use the **Parallel.Invoke** method to run several tasks in parallel.

Using the Parallel.Invoke Method

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

Running Loop Iterations in Parallel

The **Parallel** class also provides methods that you can use to run **for** and **foreach** loop iterations in parallel. Clearly it will not always be appropriate to run loop iterations in parallel. For example, if you want to compare sequential values, you must run your loop iterations sequentially. However, if each loop iteration represents an independent operation, running loop iterations in parallel enables you to maximize your use of the available processing power.

To run **for** loop iterations in parallel, you can use the **Parallel.For** method. This method has several overloads to cater to many different scenarios. In its simplest form, the **Parallel.For** method takes three parameters:

- An **Int32** parameter that represents the start index for the operation, inclusive.
- An **Int32** parameter that represents the end index for the operation, exclusive.
- An **Action<Int32>** delegate that is executed once per iteration.

The following code example shows how to use a **Parallel.For** loop. In this example, each element of an array is set to the square root of the index value. This is a simple example of a loop in which the order of the iterations does not matter.

Using a Parallel.For Loop

```
int from = 0;
int to = 500000;
double[] array = new double[capacity];
// This is a sequential implementation:
for(int index = 0; index < 500000; index++)
{
   array[index] = Math.Sqrt(index);
}
// This is the equivalent parallel implementation:
Parallel.For(from, to, index =>
{
   array[index] = Math.Sqrt(index);
});
```

To run **foreach** loop iterations in parallel, you can use the **Parallel.ForEach** method. Like the **Parallel.For** method, the **Parallel.ForEach** method includes many different overloads. In its simplest form, the **Parallel.ForEach** method takes two parameters:

- An **IEnumerable<TSource>** collection that you want to iterate over.
- An **Action<TSource>** delegate that is executed once per iteration.

The following code example shows how to use a **Parallel.ForEach** loop. In this example, you iterate over a generic list of **Coffee** objects. For each item, you call a method named **CheckAvailability** that accepts a **Coffee** object as an argument.

Using a Parallel.ForEach Loop

```
var coffeeList = new List<Coffee>();
// Populate the coffee list…
// This is a sequential implementation:
foreach(Coffee coffee in coffeeList)
{
   CheckAvailability(coffee);
}
// This is the equivalent parallel implementation:
Parallel.ForEach(coffeeList, coffee => CheckAvailability(coffee));
```

> **Additional Reading:** For more information and examples about running data operations in parallel, refer Data Parallelism (Task Parallel Library) at **http://go.microsoft.com/fwlink/?LinkID=267839**.

Using Parallel LINQ

Parallel LINQ (PLINQ) is an implementation of Language-Integrated Query (LINQ) that supports parallel operations. In most cases, PLINQ syntax is identical to regular LINQ syntax. When you write a LINQ expression, you can “opt in” to PLINQ by calling the **AsParallel** extension method on your **IEnumerable** data source.

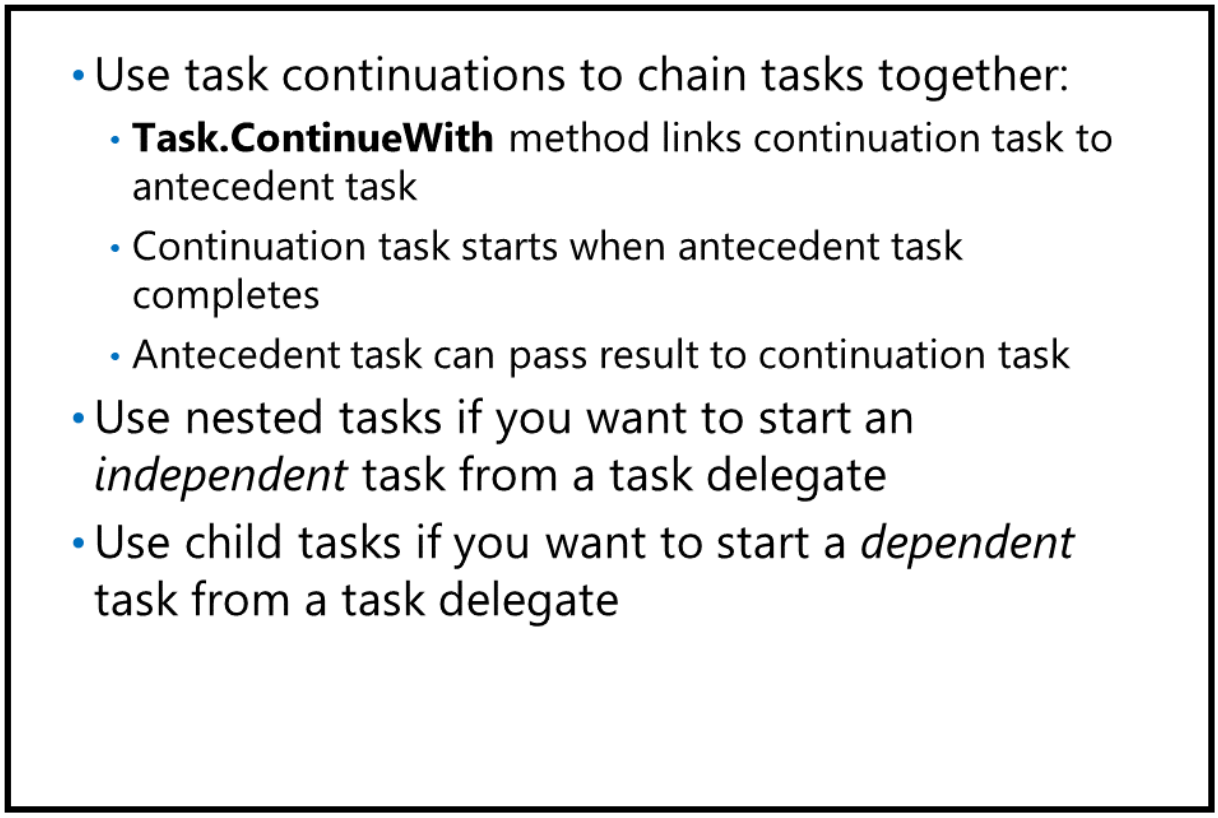
The following code example shows how to write a PLINQ query.

Using PLINQ

```
var coffeeList = new List<Coffee>();
// Populate the coffee list…
var strongCoffees =
   from coffee in coffeeList.AsParallel()
   where coffee.Strength > 3
   select coffee;
```

> **Additional Reading:** For more information about PLINQ, refer Parallel LINQ (PLINQ) at **http://go.microsoft.com/fwlink/?LinkID=267840**.

Linking Tasks



Sometimes it is useful to sequence tasks. For example, you might require that if a task completes successfully, another task is run, or if the task fails, a different task is run that possibly needs to perform some kind of recovery process. A task that runs only when a previous task finishes is called a *continuation*. This approach enables you to construct a pipeline of background operations.

Additionally, a task may spawn other tasks if the work that it needs to perform is also multithreaded in nature. The *parent* task (the task that spawned the new or *nested* tasks) can wait for the nested tasks to complete before it finishes itself, or it can return and let the child tasks continue running asynchronously. Tasks that cause the parent task to wait are called *child* tasks.

Continuation Tasks

Continuation tasks enable you to chain multiple tasks together so that they execute one after another. The task that invokes another task on completion is known as the *antecedent*, and the task that it invokes is known as the *continuation*. You can pass data from the antecedent to the continuation, and you can control the execution of

the task chain in various ways. To create a basic continuation, you use the **Task.ContinueWith** method.

The following code example shows how to create a task continuation.

Creating a Task Continuation

```
// Create a task that returns a string.
Task<string> firstTask = new Task<string>( () => return "Hello" );
// Create the continuation task.
// The delegate takes the result of the antecedent task as an
argument.
Task<string> secondTask = firstTask.ContinueWith( (antecedent) =>
   {
      return String.Format("{0}, World!", antecedent.Result));
   }
// Start the antecedent task.
firstTask.Start();
// Use the continuation task's result.
Console.WriteLine(secondTask.Result);
// Displays "Hello, World!"
```

Notice that when you create the continuation task, you are providing the result of the first task as an argument to the delegate of the second task. For example, you might use the first task to collect some data and then invoke a continuation task to process the data. Continuation tasks do not have to return the same result type as their antecedent tasks.

> **Note:** Continuations enable you to implement the Promise pattern. This is a common idiom that many asynchronous environments use to ensure that operations perform in a guaranteed sequence.

> **Additional Reading:** For more information about continuation tasks, refer Continuation Tasks at **http://go.microsoft.com/fwlink/?LinkID=267841**.

Nested Tasks and Child Tasks

A nested task is simply a task that you create within the delegate of another task. When you create tasks in this way, the nested task and the outer task are essentially independent. The outer task does not need to wait for the nested task to complete before it finishes.

A child task is a type of nested task, except that you specify the **AttachedToParent** option when you create the child task. In this case, the child task and the parent task become far more closely connected. The status of the parent task depends on the status of any child tasks—in other words, a parent task cannot complete until all of its child tasks have completed. The parent task will also propagate any exceptions that its child tasks throw.

To illustrate the difference between nested tasks and child tasks, consider these two simple examples.

The following code example shows how to create a nested task.

Creating Nested Tasks

```
var outer = Task.Run( () =>
{
   Console.WriteLine("Outer task starting…");
   var inner = Task.Run( () =>
   {
      Console.WriteLine("Nested task starting…");
      Thread.SpinWait(500000);
      Console.WriteLine("Nested task completing…");
   });
});
outer.Wait();
Console.WriteLine("Outer task completed.");
/* Output:
      Outer task starting…
      Nested task starting…
```

```
      Outer task completed.
      Nested task completing…
*/
```

In this case, due to the delay in the nested task, the outer task completes before the nested task.

The following code example shows how to create a child task.

Creating Child Tasks

```
var parent = Task.Run( () =>
{
   Console.WriteLine("Parent task starting…");
   var child = Task.Run( () =>
   {
      Console.WriteLine("Child task starting…");
      Thread.SpinWait(500000);
      Console.WriteLine("Child task completing…");
   }, TaskCreationOptions.AttachedToParent);
});
parent.Wait();
Console.WriteLine("Parent task completed.");
/* Output:
      Parent task starting…
      Child task starting…
      Child task completing…
      Parent task completed.
*/
```

Note that the preceding example is essentially identical to the nested task example, except that in this case, the child task is created by using the **AttachedToParent** option. As a result, in this case, the parent task waits for the child task to complete before completing itself.

Nested tasks are useful because they enable you to break down asynchronous operations into smaller units that can themselves be distributed across available threads. By contrast, it is more useful to use parent and child tasks when you need to control synchronization by ensuring that certain child tasks are complete before the parent task returns.

> **Additional Reading:** For more information about nested tasks and child tasks, refer Nested Tasks and Child Tasks at **http://go.microsoft.com/fwlink/?LinkID=267842**.

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.
   }
}
```

When a task throws an exception, the exception is propagated back to the thread that initiated the task (the *joining thread*). The task might be linked to other tasks through continuations or child tasks, so multiple exceptions may be thrown. To ensure that all exceptions are propagated back to the joining thread, the Task Parallel Library bundles any exceptions together in an **AggregateException** object. This object exposes all of the exceptions that occurred through an **InnerExceptions** collection.

To catch exceptions on the joining thread, you must wait for the task to complete. You do this by calling the **Task.Wait** method in a **try** block and then catching an

AggregateException exception in the corresponding **catch** block. A common exception-handling scenario is to catch the **TaskCanceledException** exception that is thrown when you cancel a task.

The following code example shows how to handle task exceptions.

Catching Task Exceptions


```
static void Main(string[] args)
{
   // Create a cancellation token source and obtain a cancellation
token.
   CancellationTokenSource cts = new CancellationTokenSource();
   CancellationToken ct = cts.Token;

   // Create and start a long-running task.
   var task1 = Task.Run(() => doWork(ct),ct);
   // Cancel the task.
   cts.Cancel();
   // Handle the TaskCanceledException.
   try
   {
      task1.Wait();
   }
   catch(AggregateException ae)
   {
      foreach(var inner in ae.InnerExceptions)
      {
         if(inner is TaskCanceledException)
         {
            Console.WriteLine("The task was cancelled.");
            Console.ReadLine();
         }
         else
         {
            // If it's any other kind of exception, re-throw it.
```

```
            throw;
         }
      }
   }
}
// Method run by the task.
private void doWork(CancellationToken token);
{
   for (int i = 0; i < 100; i++)
   {
      Thread.SpinWait(500000);
      // Throw an OperationCanceledException if cancellation was
requested.
      token.ThrowIfCancellationRequested():
   }
}
```

> **Additional Reading:** For more information about handling task exceptions, refer Exception Handling (Task Parallel Library) at **http://go.microsoft.com/fwlink/?LinkID=267843**.

Lesson 2: Performing Operations Asynchronously

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 lesson, you will learn various techniques for invoking and managing asynchronous operations.

Lesson objectives

After completing this lesson, you will be able to:

- Use the **Dispatcher** object to run code on a specific thread.
- Use the **async** and **await** keywords to run asynchronous operations.
- Create methods that are compatible with the **await** operator.
- Create and invoke callback methods.
- Use the Task Parallel Library with traditional Asynchronous Programming Model (APM) implementations.
- Handle exceptions that asynchronous operations throw.

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)));
```

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 **btnGetTim**e 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

- 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;
}
```

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;
 }
```

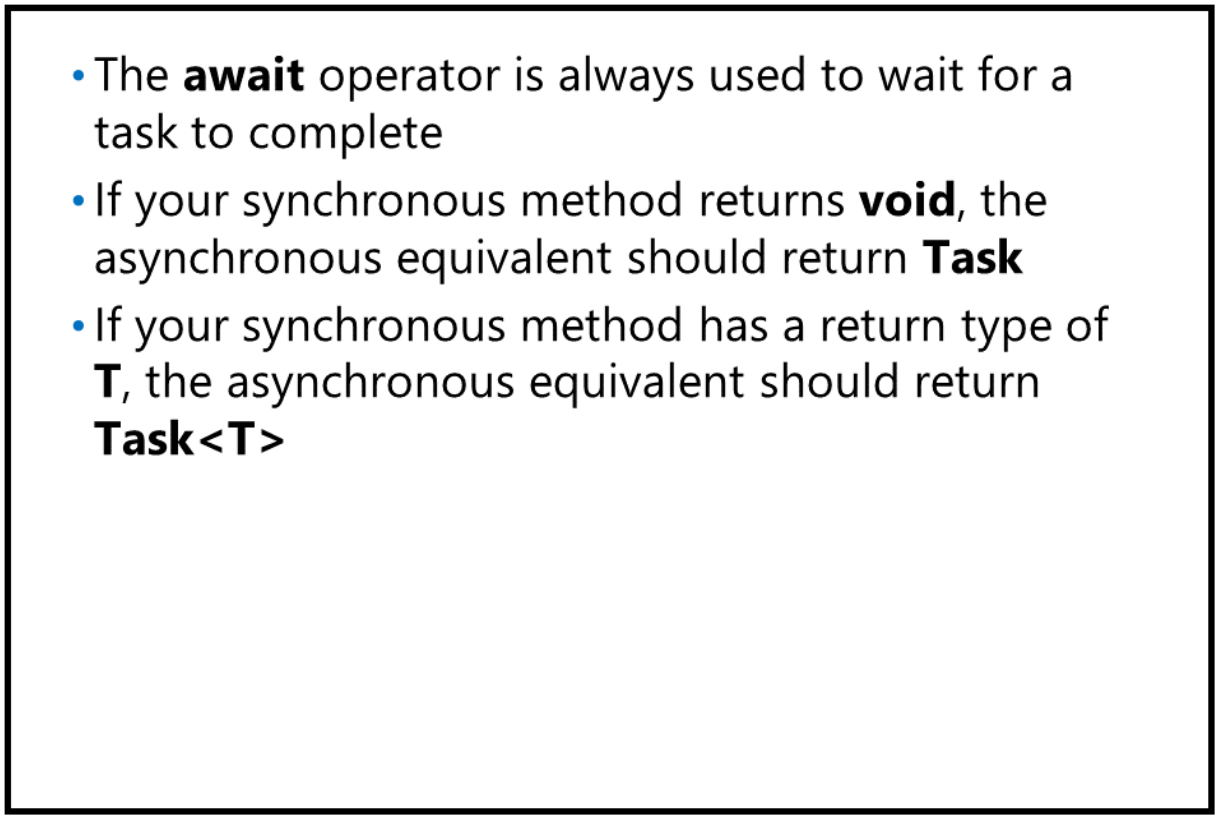
This example includes two key changes from the previous example:

- 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.

> **Additional Reading:** For more information about using **async** and **await**, refer Asynchronous Programming with Async and Await at **https://aka.ms/moc-20483c-m10-pg2**.

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.

There's a major difference between returning **void** or **Task** from an **async** method. As long as the method returns **Task,** it's a part of the **Async** operation, and you'll have to use **await** to call the method. The operation will pause until the **async** operation is complete. However, returning **void** from an **async** method will not block the method, and the next line will be called immediately. An **async void** method is the launching point of an **async** operation, even though the method itself cannot be **awaited**.

As a result, **void async** methods are usually event handlers, or a commands-execute handler, as they're the launching point of the operation.

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.

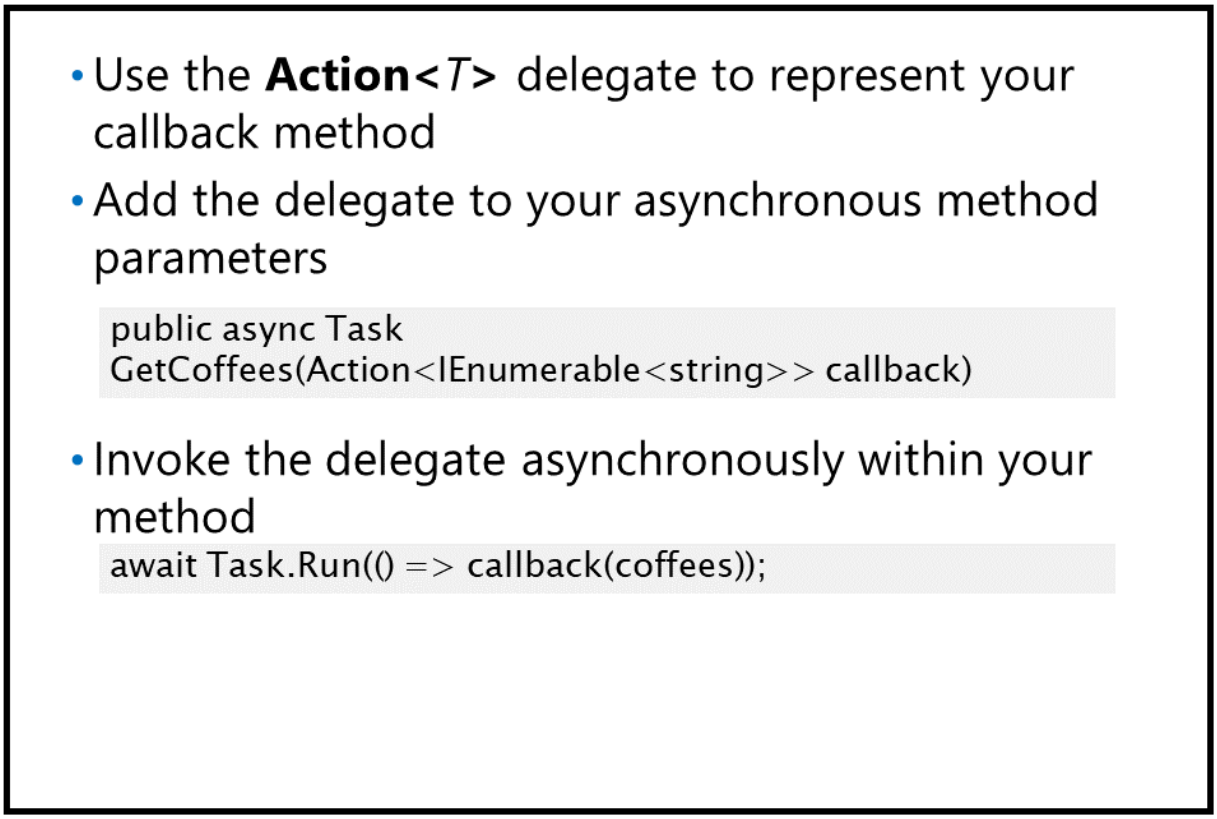
Calling an Awaitable Asynchronous Method

```
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.

> **Additional Reading:** For more information about return types for asynchronous methods, refer Async Return Types at **https://aka.ms/moc-20483c-m10-pg3**.

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

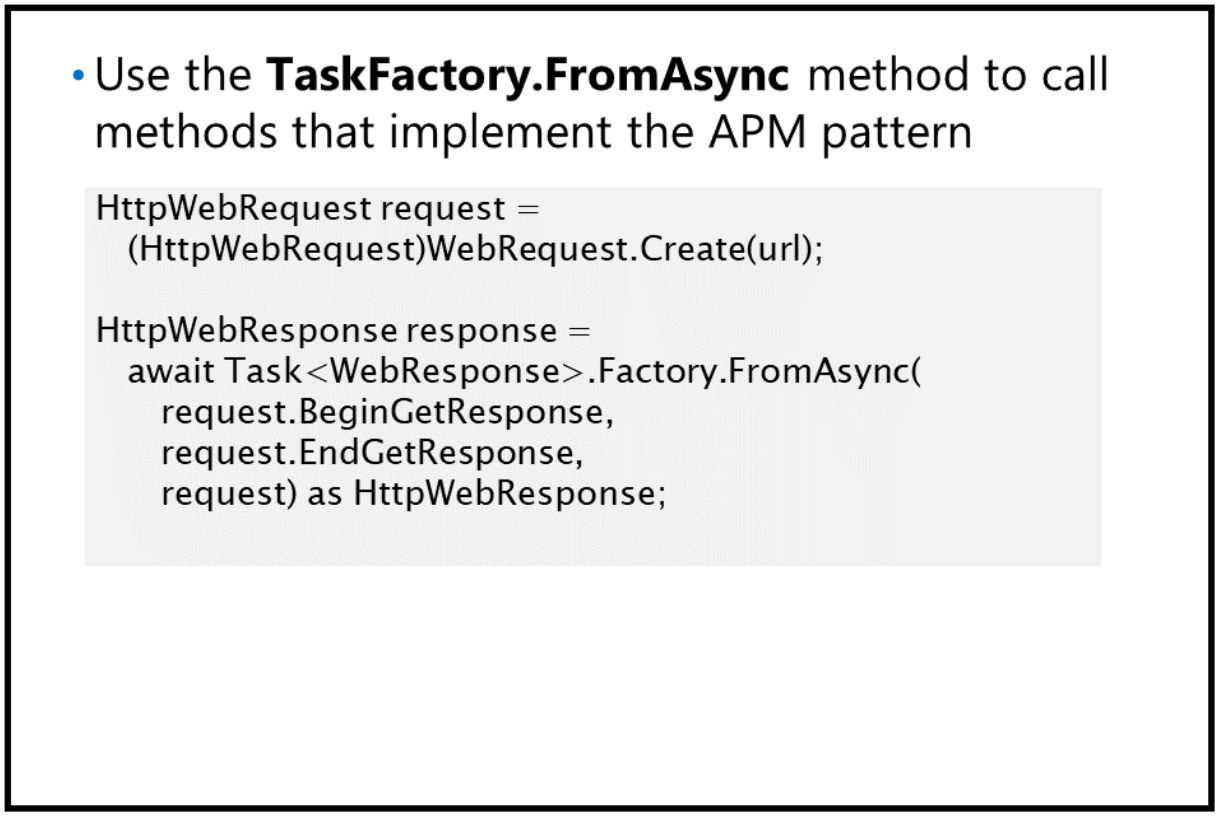
```
// This is the click event handler for the button.
private async void btnGetCoffees_Click(object sender,
RoutedEventArgs e)
{
   await GetCoffees(RemoveDuplicates);
}
// This is the asynchronous method.
public async Task GetCoffees(Action<IEnumerable<string>> callback)
{
   // Simulate a call to 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.

> **Reference Links:** For more information, refer Action<T> Delegate at **https://aka.ms/moc-20483c-m10-pg4**.

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 **Begin***OperationName* method that starts the asynchronous operation and an **End***OperationName* method that provides the results of the asynchronous operation. You typically call the **End***OperationName* 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 **Begin***OperationName* method returns an **IAsyncResult** object, and the **End***OperationName* 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 **End***OperationName* 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);
}
```

> **Additional Reading:** For more information about using the Task Parallel Library with APM patterns, refer TPL and Traditional .NET Framework Asynchronous Programming at **http://go.microsoft.com/fwlink/?LinkID=267847**.

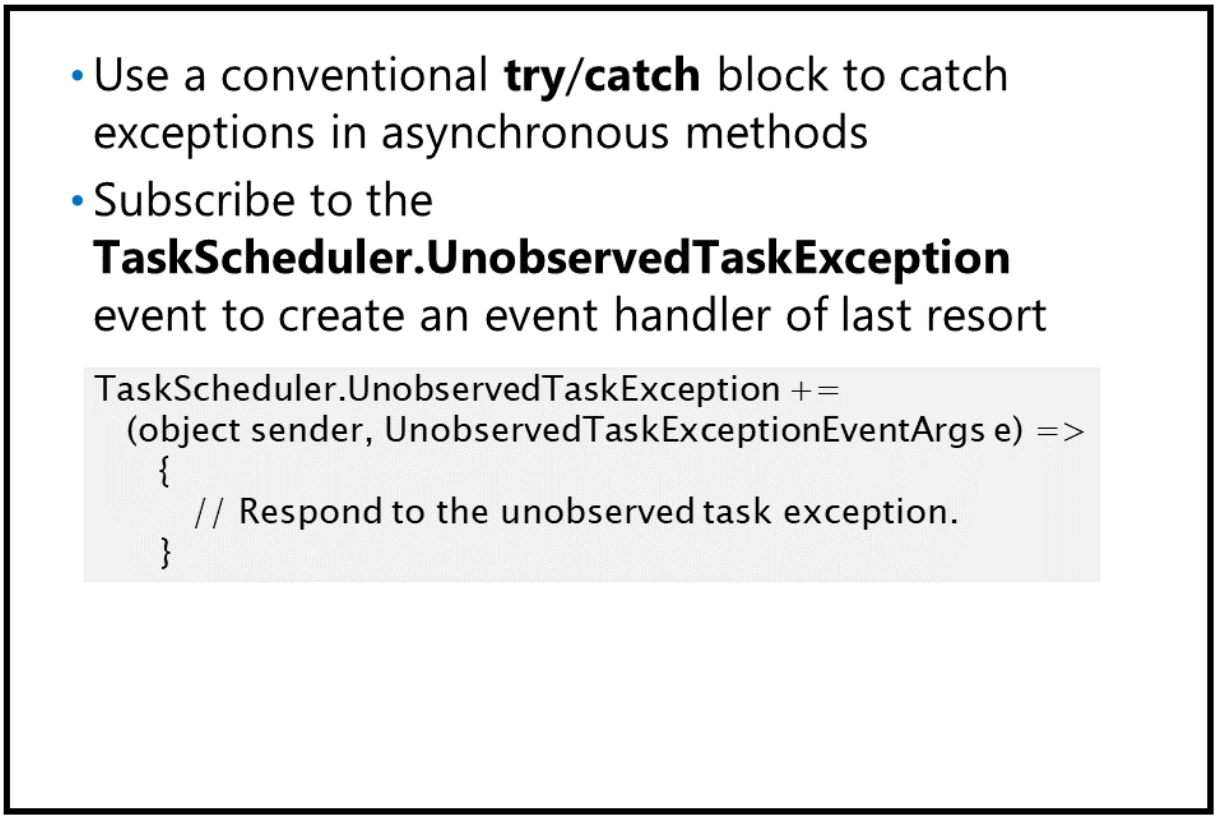
Demonstration: Using the Task Parallel Library to Invoke APM Operations

This demonstration explains how to convert an application that uses a traditional APM implementation to use the Task Parallel Library instead. The demonstration illustrates how the Task Parallel Library approach results in shorter and simpler code.

Demonstration steps

You will find the steps in the **Demonstration: Using the Task Parallel Library to Invoke APM Operations** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md.

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 notice 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.

Lesson 3: Synchronizing Concurrent Access to Data

Introducing multithreading into your applications has many advantages in terms of performance and responsiveness. However, it also introduces new challenges. When you can simultaneously update a resource from multiple threads, the resource can become corrupted or can be left in an unpredictable state.

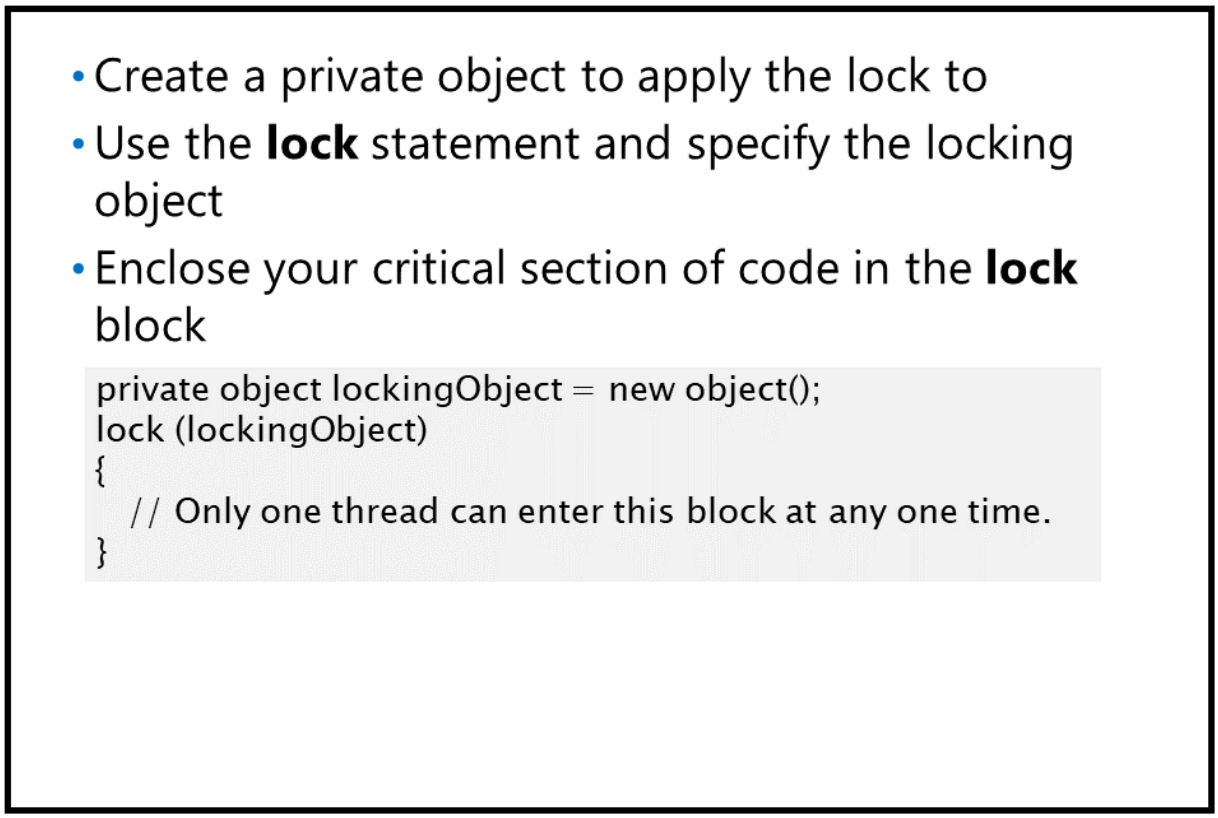
In this lesson, you will learn how to use various synchronization techniques to ensure that you access resources in a *thread-safe* manner—in other words, in a way that prevents concurrent access from having unpredictable effects.

Lesson objectives

After completing this lesson, you will be able to:

- Use **lock** statements to prevent concurrent access to code.
- Use synchronization primitives to restrict and control access to resources.
- Use concurrent collections to store data in a thread-safe way.

Using Locks



When you introduce multithreading into your applications, you can often encounter situations in which a resource is simultaneously accessed from multiple threads. If each of these threads can alter the resource, the resource can end up in an unpredictable state. For example, suppose you create a class that manages stock levels of coffee. When you place an order for a number of coffees, a method in the class will:

1. Check the current stock level.
2. If sufficient stock exists, make the coffees.

3. Subtract the number of coffees from the current stock level.

If only one thread can call this method at any one time, everything will work fine. However, suppose two threads call this method at the same time. The current stock level might change between steps 1 and 2, or between steps 2 and 3, making it impossible to keep track of the current stock level and establish whether you have sufficient stock to make a coffee.

To solve this problem, you can use the **lock** keyword to apply a mutual-exclusion lock to critical sections of code. A mutual-exclusion lock means that if one thread is accessing the critical section, all other threads are locked out. To apply a lock, you use the following syntax:

lock (** object **) {** statement block **}

The first thing to notice is that you apply the lock to an object. This is because the lock works by ensuring that only one thread can access that object at any one time. This object should be private and should serve no other role in your logic; its purpose is to provide something for the **lock** keyword to make mutually exclusive. Next, you put your critical section of code inside the **lock** block. While the **lock** statement is in scope, only one thread can enter the critical section at any one time.

The following code example shows how to use the **lock** keyword to prevent concurrent access to a block of code.

Using the Lock Keyword

```
class Coffee
{
   private object coffeeLock = new object();
   int stock;
   public Coffee(int initialStock)
   {
      stock = initialStock;
```

```
    }
    public bool MakeCoffees(int coffeesRequired)
    {
       lock (coffeeLock)
       {
          if (stock >= coffeesRequired)
          {
             Console.WriteLine(String.Format("Stock level before
making coffees: {0}", stock));
             stock = stock - coffeesRequired;
             Console.WriteLine(String.Format("{0} coffees made",
coffeesRequired));
             Console.WriteLine(String.Format("Stock level after
making coffees: {0}", stock));
             return true;
          }
          else
          {
             Console.WriteLine("Insufficient stock to make coffees");
             return false;
          }
       }
    }
}
```

In the previous example, the **lock** statement ensures that only one thread can enter the critical section of code within the **MakeCoffee** method at any one time.

> **Note:** Internally, the **lock** statement actually uses another Microsoft Visual C#® locking mechanism, the **Monitor.Enter** and **Monitor.Exit** methods, to apply a mutual exclusion lock to a critical section of code. For more information, refer lock Statement (C# Reference) at **http://go.microsoft.com/fwlink/?LinkID=267848** and Thread Synchronization (C# and Visual Basic) at **https://aka.ms/moc-20483c-m10-pg5**.

Demonstration: Using Lock Statements

This demonstration illustrates how the **lock** statement prevents concurrent access to critical sections of code. The demonstration solution contains a class named **Coffee** that includes a method named **MakeCoffees**. The **MakeCoffees** method does the following:

1. Checks the current coffee stock level.
2. Fulfills the order, if the current stock level is sufficient.
3. Subtracts the number of coffees ordered from the current stock level.

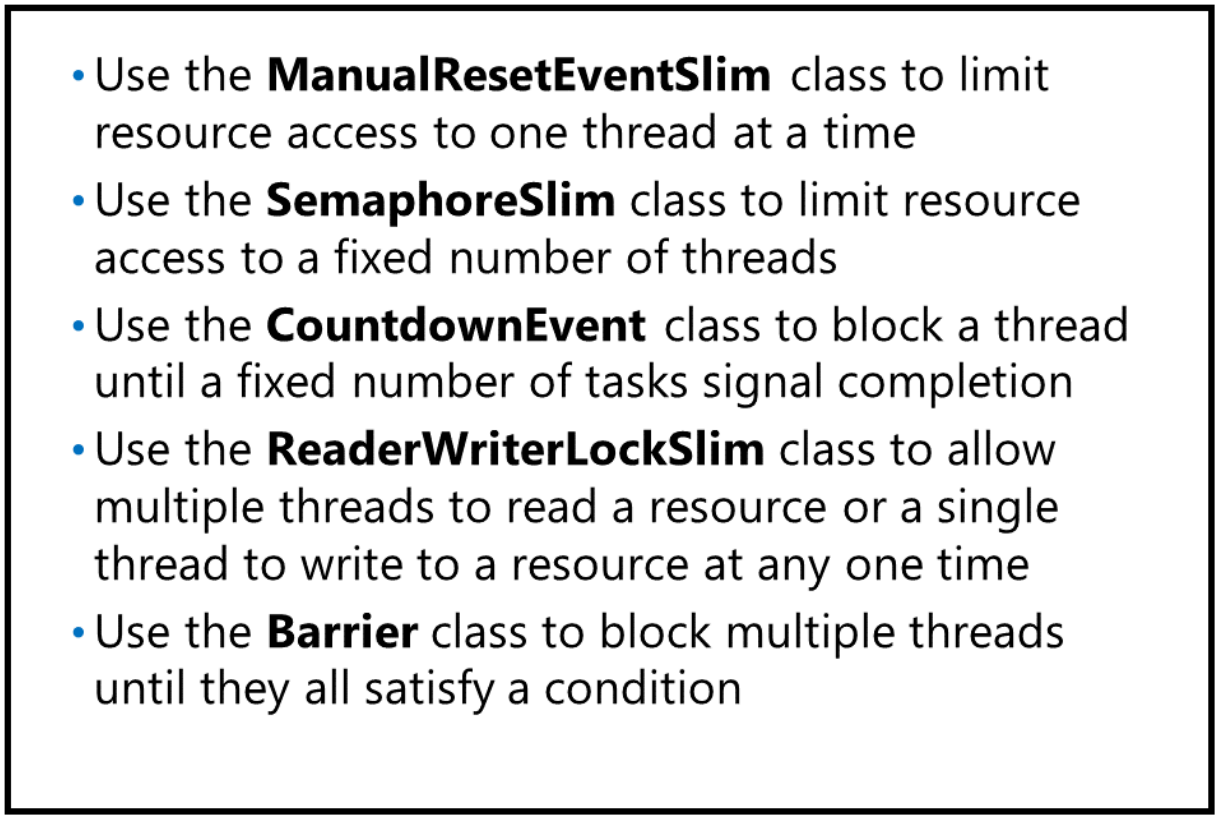
In the **Program** class, a **Parallel.For** loop creates 100 parallel operations. Each parallel operation places an order for between one and 100 coffees at random.

When you use a **lock** statement to apply a mutual-exclusion lock to the critical section of code in the **MakeCoffees** method, the method will always function correctly. However, if you remove the **lock** statement, the logic can fail, the stock level becomes negative, and the application throws an exception.

Demonstration steps

You will find the steps in the Demonstration: Using Lock Statements section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md.

Using Synchronization Primitives with the Task Parallel Library



A *synchronization primitive* is a mechanism by which an operating system enables its users, in this case the .NET runtime, to control the synchronization of threads. The Task Parallel Library supports a wide range of synchronization primitives that enable you to control access to resources in a variety of ways. These are the most commonly used synchronization primitives:

- The **ManualResetEventSlim** class enables one or more threads to wait for an event. A **ManualResetEventSlim** object can be in one of two states: *signaled* and *unsignaled*. If a thread calls the **Wait** method and the **ManualResetEventSlim** object is unsignaled, the thread is blocked until the **ManualResetEventSlim** object state changes to signaled. In your task logic, you can call the **Set** or **Reset** methods on the **ManualResetEventSlim** object to change the state to signaled or unsignaled respectively. You typically use the **ManualResetEventSlim** class to ensure that only one thread can access a resource at any one time.

> **Reference Links:** For more information about the **ManualResetEventSlim** class, refer to the ManualResetEventSlim Class page at **https://aka.ms/moc-20483c-m10-pg6**.

- The **SemaphoreSlim** class enables you to restrict access to a resource, or a pool

of resources, to a limited number of concurrent threads. The **SemaphoreSlim** class uses an integer counter to track the number of threads that are currently accessing a resource or a pool of resources. When you create a **SemaphoreSlim** object, you specify an initial value and an optional maximum value. When a thread wants to access the protected resources, it calls the **Wait** method on the **SemaphoreSlim** object. If the value of the **SemaphoreSlim** object is above zero, the counter is decremented and the thread is granted access. When the thread has finished, it should call the **Release** method on the **SemaphoreSlim** object, and the counter is incremented. If a thread calls the **Wait** method and the counter is zero, the thread is blocked until another thread calls the **Release** method. For example, if your coffee shop has three coffee machines and each can only process one order at a time, you might use a **SemaphoreSlim** object to prevent more than three threads simultaneously ordering a coffee.

> **Additional Reading:** For more information about the **SemaphoreSlim** class, refer to the SemaphoreSlim Class page at **https://aka.ms/moc-20483c-m10-pg7**.

- The **CountdownEvent** class enables you to block a thread until a fixed number of threads have signaled the **CountdownEvent** object. When you create a **CountdownEvent** object, you specify an initial integer value. When a thread completes an operation, it can call the **Signal** method on the **CountdownEvent** object to decrement the integer value. Any threads that call the **Wait** method on the **CountdownEvent** object are blocked until the counter reaches zero. For example, if you need to run ten tasks before you continue with your code, you can create a **CountdownEvent** object with an initial value of ten, signal the **CountdownEvent** object from each task, and wait for the **CountdownEvent** object to reach zero before you proceed. This is useful because your code can dynamically set the initial value of the counter depending on how much work there is to be done.

> **Additional Reading:** For more information about the **CountdownEvent** class, refer to the CountdownEvent Class page at **https://aka.ms/moc-20483c-m10-pg8**

- The **ReaderWriterLockSlim** class enables you to restrict write access to a resource to one thread at a time, while permitting multiple threads to read from the resource simultaneously. If a thread wants to read the resource, it calls the **EnterReadLock** method, reads the resource, and then calls the **ExitReadLock** method. If a thread wants to write to the resource, it calls the **EnterWriteLock** method. If one or more threads have a read lock on the resource, the **EnterWriteLock** method blocks until all read locks are released. When the thread has finished writing to the resource, it calls the **ExitWriteLock** method. Calls to the **EnterReadLock** method are blocked until the write lock is released. As a result, at any one time, a resource can be locked by either one writer or multiple readers. This type of read/write lock is useful in a wide range of scenarios. For example, a banking application might permit multiple threads to read an account balance simultaneously, but a thread that wants to modify the account balance requires an exclusive lock.

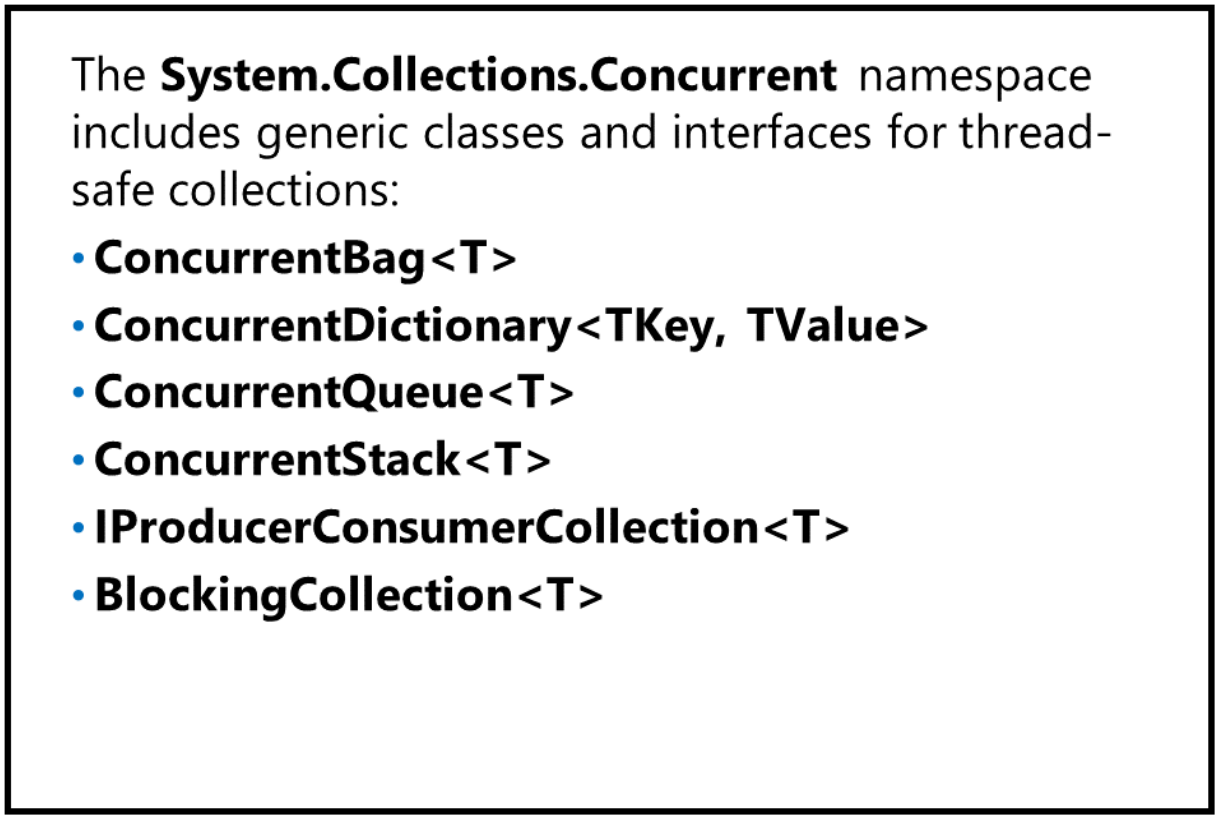
> **Additional Reading:** For more information about the **ReaderWriterLockSlim** class, refer to the ReaderWriterLockSlim Class page at **https://aka.ms/moc-20483c-m10-pg9**.

- The **Barrier** class enables you to temporarily halt the execution of several threads until they have all reached a particular point. When you create a **Barrier** object, you specify an initial number of participants. You can change this number at a later date by calling the **AddParticipant** and **RemoveParticipant** methods. When a thread reaches the synchronization point, it calls the **SignalAndWait** method on the **Barrier** object. This decrements the **Barrier** counter and also blocks the calling thread until the counter reaches zero. When the counter reaches zero, all threads are allowed to continue. The **Barrier** class is often used in forecasting scenarios, where various tasks perform interrelated calculations on one time window and then wait for all of the other tasks to reach the same point before performing interrelated calculations on the next time window.

> **Additional Reading:** For more information about the **Barrier** class, refer to the Barrier Class page at **https://aka.ms/moc-20483c-m10-pg10**.

Many of these classes enable you to set timeouts in terms of the number of *spins*. When a thread is waiting for an event, it spins. The length of time a spin takes depends on the computer that is running the thread. For example, if you use the **ManualResetEventSlim** class, you can specify the maximum number of spins as an argument to the constructor. If a thread is waiting for the **ManualResetEventSlim** object to signal and it reaches the maximum number of spins, the thread is suspended and stops using processor resources. This helps to ensure that waiting tasks do not consume excessive processor time.

Using Concurrent Collections



The standard collection classes in the .NET Framework are, by default, not thread-safe. When you access collections from tasks or other multithreading techniques, you must ensure that you do not compromise the integrity of the collections. One way to do this is to use the synchronization primitives described earlier in this lesson to control concurrent access to your collections. However, the .NET Framework also includes a set of collections that are specifically designed for thread-safe access. The following table describes some of the classes and interfaces that the **System.Collections.Concurrent** namespace provides.

Class or interface	Description

Class or interface	Description
ConcurrentBag<T>	This class provides a thread-safe way to store an unordered collection of items.
ConcurrentDictionary<TKey, TValue>	This class provides a thread-safe alternative to the **Dictionary<TKey, TValue>** class.
ConcurrentQueue<T>	This class provides a thread-safe alternative to the **Queue<T>** class.
ConcurrentStack<T>	This class provides a thread-safe alternative to the **Stack<T>** class.
IProducerConsumerCollection<T>	This interface defines methods for implementing classes that exhibit producer/consumer behavior; in other words, classes that distinguish between producers who add items to a collection and consumers who read items from a collection. This distinction is important because these collections need to implement a read/write locking pattern, where the collection can be locked either by a single writer or by multiple readers. The **ConcurrentBag<T>**, **ConcurrentQueue<T>**, and **ConcurrentStack<T>** classes all implement this interface.
BlockingCollection<T>	This class acts as a wrapper for collections that implement the **IProducerConsumerCollection<T>** interface. For example, it can block read requests until a read lock is available, rather than simply refusing a request if a lock is unavailable. You can also use the **BlockingCollection<T>** class to limit the size of the underlying collection. In this case, requests to add items are blocked until space is available.

Consider an order management system for Fourth Coffee that uses a **ConcurrentQueue<T>** object to represent the queue of orders that customers have placed. Orders are added to the queue at a single order point, but they can be picked up by one of three baristas working in the store. The following example simulates this scenario by creating one task that places an order every 0.25 seconds and three tasks that pick up orders as they become available.

The following code example shows how to use the **ConcurrentQueue<T>** class to queue and de-queue objects in a thread-safe way.

Using the ConcurrentQueue<T> Collection

```
class Program
{
   static ConcurrentQueue<string> queue = new
ConcurrentQueue<string>();
   static void PlaceOrders()
   {
      for (int i = 1; i <= 100; i++)
      {
         Thread.Sleep(250);
         String order = String.Format("Order {0}", i);
         queue.Enqueue(order);
         Console.WriteLine("Added {0}", order);
      }
   }
   static void ProcessOrders()
   {
      while (true) //continue indefinitely
      if (queue.TryDequeue(out order))
      {
         Console.WriteLine("Processed {0}", order);
      }
   }
   static void Main(string[] args)
   {
      var taskPlaceOrders = Task.Run(() => PlaceOrders());
      Task.Run(() => ProcessOrders());
      Task.Run(() => ProcessOrders());
      Task.Run(() => ProcessOrders());
      taskPlaceOrders.Wait();
      Console.WriteLine("Press ENTER to finish");
      Console.ReadLine();
   }
}
```

Reference Links: For more information about using concurrent collections, refer System.Collections.Concurrent Namespace at **https://aka.ms/moc-**

20483c-m10-pg11.

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.

Demonstration steps

You will find the steps in the “Demonstration: Improving the Responsiveness and Performance of the Application Lab.“ section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md.

Lab: Improving the Responsiveness and Performance of the Application

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.

Objectives

After completing this lab, you will be able to:

- Use the **async** and **await** keywords to implement asynchronous methods.
- Use events and user controls to provide visual feedback during long-running operations.

Lab setup

Estimated Time: 75 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_LAB_MANUAL.md**.

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_LAK.md**.

Exercise 1: Ensuring That the UI Remains Responsive When Retrieving Teacher Data

---

Scenario

In this exercise, you will modify the functionality that retrieves data for teachers to make use of asynchronous programming techniques. First, you will modify the code that gets the details of the current user (when the user is a teacher) to run asynchronously. You will use an asynchronous task to run the LINQ query and use the **await** operator to return the results of the query. Next, you will modify the code that retrieves the list of students for a teacher. In this case, you will configure the code that retrieves the list of students to run asynchronously. When the operation is complete, your code will invoke a callback method to update the UI with the list of students. Finally, you will build and test the application and verify that the UI remains responsive while the application is retrieving data.

> **Result**: After completing this exercise, you should have updated the Grades application to retrieve data asynchronously.

Exercise 2: Providing Visual Feedback During Long-Running Operations

---

Scenario

In this exercise, you will create a user control that displays a progress indicator while the Grades application is retrieving data. You will add this user control to the main page but will initially hide it from view. Next, you will modify the code that retrieves data so that it raises one event when the data retrieval starts and another event when the data retrieval is complete. You will create handler methods for these events that toggle the visibility of the progress indicator control, so that the application displays the progress indicator when data retrieval starts and hides it when data retrieval is complete. Finally, you will build and test the application and verify that the UI displays the progress indicator while the application is retrieving data.

> **Result**: After completing this exercise, you should have updated the Grades application to display a progress indicator while the application is retrieving data.

Module review and takeaways

In this module, you have learned a variety of asynchronous programming techniques for Visual C#, including how to use the Task Parallel Library, how to use the **async** and **await** keywords, and how to use synchronization primitives.

Review Question(s)

Check Your Knowledge

Select the best answer

You create and start three tasks named task1, task2, and task3. You want to block the joining thread until all of these tasks are complete. Which code example should you use to accomplish this?

task1.Wait();task2.Wait();task3.Wait();

Task.WaitAll(task1, task2, task3);

Task.WaitAny(task1, task2, task3);

Task.WhenAll(task1, task2, task3);

Task.WhenAny(task1, task2, task3);

Check answer Show solution Reset

Check Your Knowledge

Select the best answer

You have a synchronous method with the following signature:

```
public IEnumerable<string> GetCoffees(string country, int strength)
```

You want to convert this method to an asynchronous method. What should the signature of the asynchronous method be?

public async IEnumerable<string> GetCoffees(string country, int strength)

public async Task<string> GetCoffees(string country, int strength)

public async Task<IEnumerable<string>> GetCoffees(string country, int strength)

public async Task GetCoffees(string country, int strength, out string result)

public async Task GetCoffees(string country, int strength, out IEnumerable<string> result)

Check answer Show solution Reset

Check Your Knowledge

Select the best answer

You want to ensure that no more than five threads can access a resource at any one time. Which synchronization primitive should you use?

The ManualResetEventSlim class.

The SemaphoreSlim class.

The CountdownEvent class.

The ReaderWriterLockSlim class.

The Barrier class.

Check answer Show solution Reset