

Microsoft® Official Course



Module 10

Improving Application Performance and Responsiveness

Microsoft®

Brief intro to Lambda Expressions

- A **lambda expression** is a **shorthand syntax** that provides a simple and concise way to **define anonymous delegates**, that can take parameters and return a result
- A **lambda expression** is a **block of code that is treated as an object**.
 - It can be passed as an argument to methods, and it can also be returned by method calls
 - See also <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions#code-try-0>
- It has the following form:
 - **(input parameters) => expression**
 - 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 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.
- **Action** (Lambda expressions that don't return a value) vs **Func** (do return a value) ...

```
Func<int, int> square = x => x * x;  
Console.WriteLine(square(5));  
// Output:  
// 25
```

```
int[] numbers = { 2, 3, 4, 5 };  
var squaredNumbers = numbers.Select(x => x * x);  
Console.WriteLine(string.Join(" ", squaredNumbers));  
// Output:  
// 4 9 16 25
```

```
Action act = new Action(Met1);  
Action<int> act2 = new Action<int>(Met2);  
}  
  
private static void Met1()  
{  
}  
  
private static void Met2(int x)  
{  
}
```

Using lambda expressions to Sort a List<>

```
using System;
using System.Collections.Generic;

namespace CSC340._515_BST
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Student> myList = new List<Student>();

            myList.Add(new Student("Alice", "CS"));
            myList.Add(new Student("Charlie", "MSSA"));
            myList.Add(new Student("Bob", "undecided"));

            myList.Sort((x, y) => { return x.Major.CompareTo(y.Major); });

            foreach(Student st in myList )
                Console.WriteLine(st.Name+" : "+st.Major);
        }

        class Student
        {
            public string Name { get; set; }
            public string Major { get; set; }

            public Student(string nm, string mj)
            {
                Name = nm;
                Major = mj;
            }
        }
    }
}
```

Rationale ...

- Create a WPF application and in it a method that takes some time to complete
 - Use: **Thread.Sleep(10000);**//pause for 10 seconds
- How is the user experience?

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

- The .NET Framework includes the **Task Parallel Library**.
 - a set of classes that makes it easy to distribute your code execution across **multiple threads**.
- 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**.
 - A **Task** object **runs a block of code**, and you specify this code as a **parameter to the constructor**.

Creating Tasks

- You can provide code in a method & create an **Action delegate** that wraps this method
 - A **delegate** provides a mechanism for referencing a block of code or a method.
 - Action** class is a type that enables you to **convert a method into a delegate**.
 - The **method cannot return a value**, but it can take parameters.
 - The .NET Framework Class Library **Func** class, enables you to define a **delegate that can return a result**.

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

- If the method is not reused anywhere else then use an **anonymous method**
 - An **anonymous method** is a method without a name
 - use the **delegate** keyword to **convert an anonymous method into a delegate**.

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

- Use **lambda expressions**
 - by far **the most commonly used mechanism for defining anonymous delegates**.
 - Named method**
 - Anonymous method**

```
Task task3 = new Task(() => GetTimeNow());
```

```
Task task4 = new Task(() => { Console.WriteLine("time now is: {0}", DateTime.Now); });
//this is equivalent to Task task2 = new Task(delegate { Console.WriteLine("time now is: {0}", DateTime.Now); });
```


Controlling Task Execution

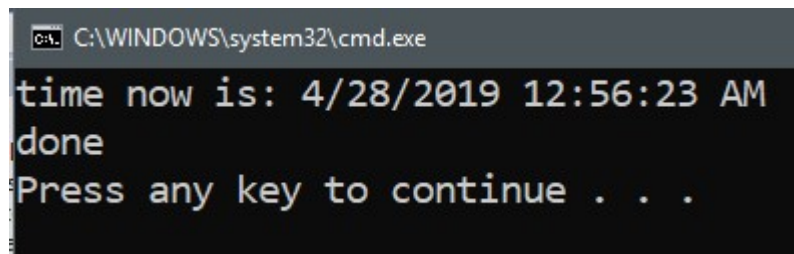
- The **task runs on a separate thread**, so your code does not need to wait for the task to complete.
 - Instead, the task and the code that invoked the task continue to run in parallel.
- To **start** a task:
 - **Task.Start** instance method
 - `task1.Start();`
 - **Task.Factory.StartNew** static method ← accepts a wide range of parameters
 - `var task3 = Task.Factory.StartNew(() => Console.WriteLine("Task 3 has completed."));`
 - **Task.Run** static method ← to queue some code with the default scheduling options
 - `var task4 = Task.Run(() => Console.WriteLine("Task 4 has completed. "));`
- To **wait** for tasks to complete:
 - **Task.Wait** instance method ← wait for a single task to finish executing
 - `task1.Wait();`
 - **Task.WaitAll** static method ← wait for multiple tasks to finish executing
 - `Task.WaitAll(tasks);` //tasks is an array of Tasks
 - **Task.WaitAny** static method ← wait for any one of a collection of tasks to finish exec.
 - `Task.WaitAny(tasks);`

Controlling Task Execution

- Compare ...

```
static void Main(string[] args)
{
    //Task task1 = new Task(new Action(GetTimeNow));
    task1.Start();
    task1.Wait();
    Console.WriteLine("done");
}
```

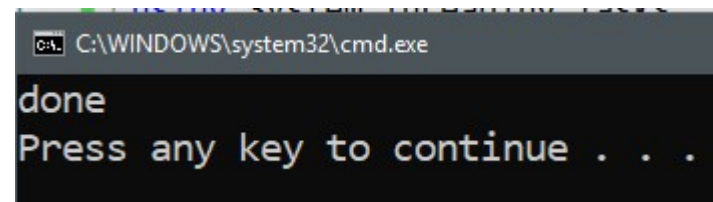
```
static Task task1 = new Task(new Action(GetTimeNow));
private static void GetTimeNow()
{
    Console.WriteLine("time now is: {0}", DateTime.Now);
}
```



```
C:\WINDOWS\system32\cmd.exe
time now is: 4/28/2019 12:56:23 AM
done
Press any key to continue . . .
```

```
static void Main(string[] args)
{
    //Task task1 = new Task(new Action(GetTimeNow));
    task1.Start();
    //task1.Wait();
    Console.WriteLine("done");
}
```

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



```
C:\WINDOWS\system32\cmd.exe
done
Press any key to continue . . .
```

Returning a Value from a Task

- The regular **Task** class does not enable you to return a value ...
- Use the **Task<TResult>** class
 - Specify the return type in the type argument
- Get the result from the **Result** (read-only) property
 - If you access the Result property before the task has finished running, your code will wait until a result is available before proceeding.

```
// 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);
```

Cancelling Long-Running Tasks

Tasks are often used to perform **long-running operations** without blocking the UI thread, because of their asynchronous nature.

- Sometimes you want to allow the user to cancel a task

The **cancellation process** works as follows:

- When you create a task, create a **cancellation token**.
- **Pass that token** as an argument to the delegate method
- **Request cancellation** from the joining thread by calling the **Cancel** method on the cancellation token source.
- 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.
 - For example, if your task logic iterates over a collection, you might check for cancellation after each iteration.
 - If you cancel a task by returning the task method, as shown in the previous example, the task status is set to **RanToCompletion**
 - Hence it has no way of knowing why the method returned
- Return or throw an **OperationCanceledException** exception ... **must pass a second parameter ct to Run...**
 - If you want to **cancel a task** and be able to **confirm that it was cancelled**
 - To check whether a cancellation was requested and throw an **OperationCanceledException** exception if it was, you call the **ThrowIfCancellationRequested** method on the cancellation token
 - ... sets the task status to **Canceled**

```
// 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.
    ...
}
```

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation>

```
// 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) ,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.
    ...
}
```

Running Tasks in Parallel

- The static class **Parallel** provides a range of methods that you can use to **execute tasks simultaneously**.
- Use **Parallel.Invoke** to **run multiple tasks simultaneously**
 - You do not need to explicitly create each task
- Use **Parallel.For** to run **for** loop iterations in parallel
 - if each loop iteration represents an **independent operation**, then running loop iterations in parallel enables you to maximize your use of the available processing power.
 - Many different overloads available ...
 - Parameters in here ...
 - 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.
- Use **Parallel.ForEach** to run **foreach** loop iterations in parallel. Parameters in here...
 - An **IEnumerable<TSource>** collection that you want to iterate over
 - An **Action<TSource>** delegate that is executed once per iteration.
- Use **PLINQ** to run LINQ expressions in **parallel**
 - "opt in" to PLINQ by calling the **AsParallel** extension method

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

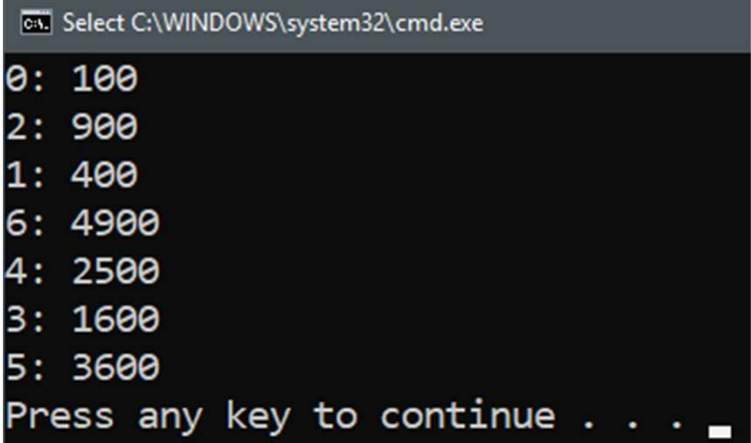
```
int from = 0;  
int to = 500000;  
double[] array = new double[capacity];  
// This is a sequential implementation:  
for(int index = 0; index < 500000; index++)  
{  
    array[index] = Math.Sqrt(index);  
}  
// This is the equivalent parallel implementation:  
Parallel.For(from, to, index => 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

```
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));
```

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

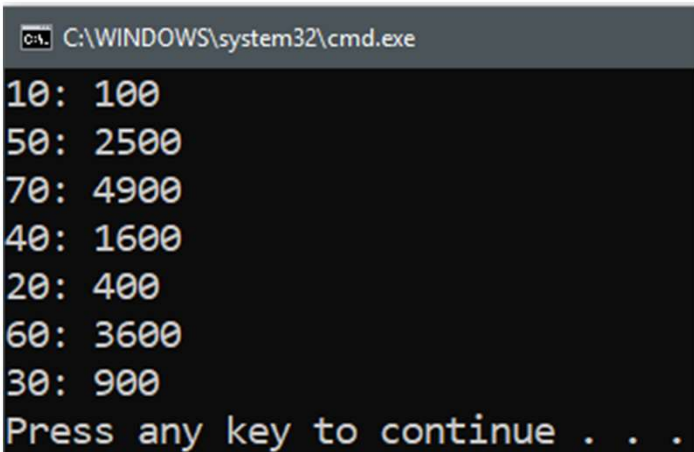
Running Tasks in Parallel

```
double[] arr = { 10, 20, 30, 40, 50, 60, 70 };  
Parallel.For(0, arr.Length, i => Console.WriteLine("{0}: {1}", i, arr[i] * arr[i]));
```



```
C:\> Select C:\WINDOWS\system32\cmd.exe  
0: 100  
2: 900  
1: 400  
6: 4900  
4: 2500  
3: 1600  
5: 3600  
Press any key to continue . . .
```

```
double[] arr = { 10, 20, 30, 40, 50, 60, 70 };  
Parallel.ForEach(arr, val => Console.WriteLine("{0}: {1}", val, val * val));
```



```
C:\> C:\WINDOWS\system32\cmd.exe  
10: 100  
50: 2500  
70: 4900  
40: 1600  
20: 400  
60: 3600  
30: 900  
Press any key to continue . . .
```


Linking 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**.
- 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

```
// 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!"
```

- Use **nested tasks** if you want to start an **independent** task from a task delegate
 - The outer task does not need to wait for the nested task to complete before it finishes.

```
var outer = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Nested task starting...");
        Thread.Sleep(500000);
        Console.WriteLine("Nested task completing...");
    });
});
outer.Wait();
Console.WriteLine("Outer task completed.");
```

```
C:\WINDOWS\system32\cmd.exe
Outer task starting.
Outer task completed.
Nested task starting.
Press any key to continue . . .
```

- Use **child tasks** if you want to start a **dependent** task from a task delegate
 - A child task is a type of **nested task**, except that you **specify the AttachedToParent option** when you create the child task.
 - a parent task cannot complete until all of its child tasks have completed

```
C:\WINDOWS\system32\cmd.exe
Outer task starting.
Nested task starting.
Nested task completing.
Outer task completed.
```

```
var outer = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Nested task starting...");
        Thread.Sleep(500000);
        Console.WriteLine("Nested task completing...");
    }, TaskCreationOptions.AttachedToParent);
});
outer.Wait();
Console.WriteLine("Outer task completed.");
```

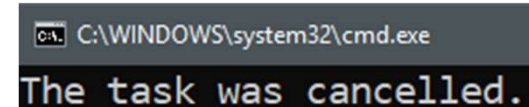
Handling Task Exceptions

- 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.
- Call **Task.Wait** to catch propagated exceptions
 - To catch exceptions on the joining thread, you must wait for the task to complete
 - A common exception handling scenario is to catch the **TaskCanceledException** exception that is thrown when you cancel a task.
- 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.
    }
}
```


Handling Task Exceptions

```
// 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;
        }
    }
}
```



C:\WINDOWS\system32\cmd.exe
The task was cancelled.

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

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

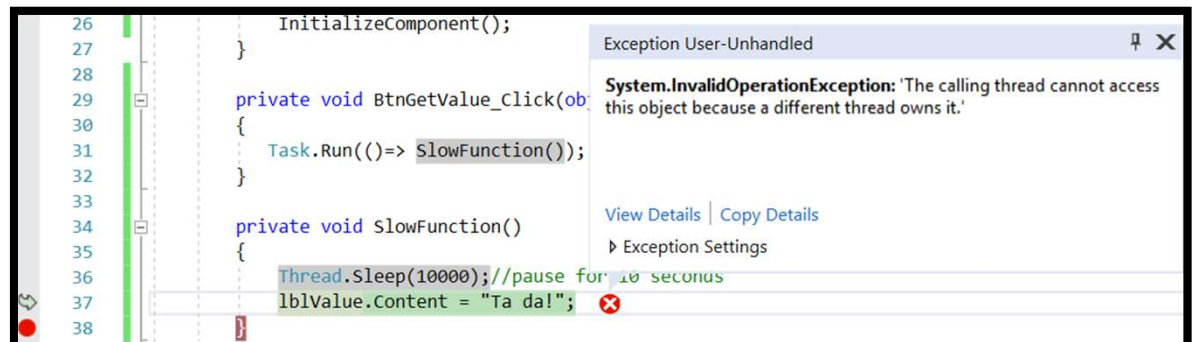
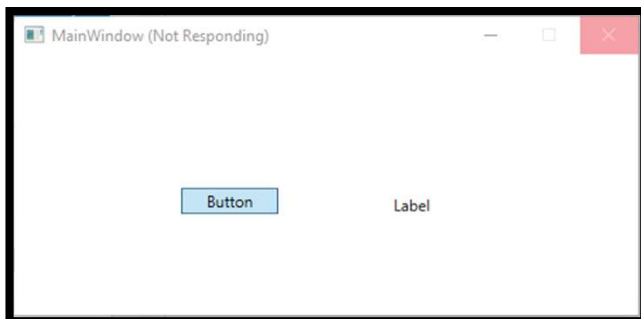
- An **asynchronous operation** is an operation that runs on a separate thread;
 - the thread that initiates an asynchronous operation **does not need to wait** for that operation to complete before it can continue
- In the .NET Framework, **each thread is associated with a Dispatcher** object.
 - The **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**
 - For example:
 - if a user clicks a button in a Windows Presentation Foundation (WPF) applic, 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**.
 - **BeginInvoke** executes a specified delegate asynchronously with specified arguments **on the thread that the Dispatcher was created on**
- **[NOTE: all UI elements are created - and therefore “owned” - by the UI thread]**

An example ...

- Here is an example:
 - V1: you have a slow function ... `SlowFunction();`
 - V2: you use a task ... `Task.Run(() => SlowFunction());`
 - You may get an **System.InvalidOperationException** ...
 - V3: Use the **Dispatcher** ...

```
public MainWindow()  
{  
    InitializeComponent();  
}  
  
private void BtnGetValue_Click(object sender, RoutedEventArgs e)  
{  
    SlowFunction();  
}  
  
private void SlowFunction()  
{  
    Thread.Sleep(10000); //pause for 10 seconds  
    lblValue.Content = "Ta da!";  
}
```

```
private void BtnGetValue_Click(object sender, RoutedEventArgs e)  
{  
    //this is an event handler ... hence it must return void!  
    Task.Run(() => SlowFunction());  
}  
  
private void SlowFunction()  
{  
    Thread.Sleep(10000); //pause for 10 seconds  
    lblValue.Content = "Ta da!";  
}
```



An example ...

- Here is an example:
 - V1: you have a slow function ... `SlowFunction();`
 - V2: you use a task ... `Task.Run(() => SlowFunction());`
 - You may get an **System.InvalidOperationException** ...
 - V3: Use the **Dispatcher** ...

```
private void BtnGetValue_Click(object sender, RoutedEventArgs e)
{
    //this is an event handler ... hence it must return void!
    Task.Run(() => SlowFunction());
}

private void SlowFunction()
{
    Thread.Sleep(10000); //pause for 10 seconds
    lblValue.Dispatcher.BeginInvoke(new Action(() => lblValue.Content = "Ta da!"));
}
```

Using the Dispatcher

- Example:

- A simple WPF application that consists of a button named **btnGetTime** and a label named **lblTime**. When the user clicks the button, one could try to use a **task** to get the current time
 - OR: your applications might retrieve data from web services or databases in response to activity on the UI
- In this example (see right)
 - an **InvalidOperationException** exception with the message "The calling thread cannot access this object because a different thread owns it"
 - because **SetTime** method is running on a **background** thread, but the **lblTime** label was created by the **UI** thread

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;
}
```

- 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
 - **BeginInvoke** method will not accept an anonymous delegate.

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;
}
```


Using **async** and **await**

- Add the **async** modifier to indicate that a method is **asynchronous**.
- Use the **await** operator within **async** methods to wait for a task to complete without blocking the thread
 - While the method is suspended at an await point, the thread that invoked the method can do other work.
 - the **async** and **await** keywords enable you to run logic asynchronously **on a single thread**.
 - Notice that when you use the **await** operator, you do not await the result of the task—you await the task itself.

Running a Synchronous Operation on the UI Thread

```
private void BtnGetValue_Click(object sender, RoutedEventArgs e)
{
    lblValue.Content = "Commencing long-running operation...";
    Task<string> task1 = Task.Run<string>(() =>
    {
        // This represents a long-running operation.
        Thread.Sleep(10000); // pause for 10 seconds
        return "Operation Complete";
    });
    // This statement blocks the UI thread until the task result is available.
    lblValue.Content = task1.Result;
}
```

Running an Asynchronous Operation on the UI Thread

```
private async void BtnGetValue_Click(object sender, RoutedEventArgs e)
{
    lblValue.Content = "Commencing long-running operation...";
    Task<string> task1 = Task.Run<string>(() =>
    {
        // This represents a long-running operation.
        Thread.Sleep(10000); // pause for 10 seconds
        return "Operation Complete";
    });
    // This statement blocks the UI thread until the task result is available.
    lblValue.Content = await task1;
}
```

Side notes ...

- The **async** and **await** keywords don't cause additional threads to be created.
- the **async** and **await** enable you to run logic **asynchronously on a single thread**.
 - Note that when you use the **await** operator, **you do not await the result of the task—you await the task itself**.
 - Use the **await** operator within **async** methods to wait for a task to complete without blocking the thread

```
private void BtnGetValue_Click(object sender, RoutedEventArgs e)
{
    lblValue.Content = "Commencing long-running operation...";
    Task<string> task1 = Task.Run<string>(() =>
    {
        // This represents a long-running operation.
        Thread.Sleep(10000); // pause for 10 seconds
        return "Operation Complete";
    });
    // This statement blocks the UI thread until the task result is available.
    lblValue.Content = task1.Result;
}
```

```
private async void BtnGetValue_Click(object sender, RoutedEventArgs e)
{
    lblValue.Content = "Commencing long-running operation...";
    Task<string> task1 = Task.Run<string>(() =>
    {
        // This represents a long-running operation.
        Thread.Sleep(10000); // pause for 10 seconds
        return "Operation Complete";
    });
    // This statement blocks the UI thread until the task result is available.
    lblValue.Content = await task1;
}
```

An **async** method runs synchronously until it reaches its first **await** expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the **async** keyword modifies doesn't contain an **await** expression or statement, the method executes synchronously. A compiler warning alerts you to any **async** methods that don't contain **await** statements, because that situation might indicate an error. See [Compiler Warning \(level 1\) CS4014](#).

Creating Awaitable Methods

- The **await** operator is always used to await the completion of a **Task** instance in a non-blocking manner
- To convert a synchronous method to an asynchronous method:
 - If your synchronous method returns **void**, the asynchronous method should return a **Task** object. The method body still should not include a **return** statement.
 - If your synchronous method has a return type of **T**, your asynchronous method should return a **Task<T>** object.

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;
}
```

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;
}
```

Calling an Awaitable Asynchronous Method

```
private async void btnGetData_Click(object sender, RoutedEventArgs e)
{
    lblResult.Content = await GetData();
}
```

- Note: all event handlers must return **void**.
 - Hence you cannot return a **Task** object from an asynchronous event handler.

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
- Use the **Action<T>** delegate to represent your callback method
 - Add the delegate to your asynchronous method parameters (T is type of callback met. argument)

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

- Invoke the delegate asynchronously within your method

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

Example below: 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.

```
// 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;  
    })  
    )  
}
```

Working with APM Operations

- Many .NET Framework classes that support asynchronous operations do so by implementing a **design pattern** known as **Asynchronous Programming Model (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 the methods
 - **BeginGetResponse** method **submits an asynchronous request** to an Internet or intranet resource
 - **EndGetResponse** method **returns the actual response** that the Internet resource provides.
- 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**.

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

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

Handling Exceptions from Awaitable Methods

- Use a conventional **try/catch** block to catch exceptions in asynchronous methods
- Example: how to catch an exception that an **awaitable** method has thrown
 - 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.

```
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;
        }
    }
}
```

Handling Exceptions from Awaitable Methods

- 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.
 - the **UnobservedTaskException** event is fired when the GC runs.
 - 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.
- 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.

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

Text Continuation

```
private async void btnCheckUrl_Click(object sender, RoutedEventArgs e)
{
    string url = txtUrl.Text;
    if (!String.IsNullOrEmpty(url))
    {
        try
        {
            HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
            //request.BeginGetResponse(new AsyncCallback(ResponseCallback), request);
            HttpWebResponse response = await Task<WebResponse>.Factory.FromAsync(request.BeginGetResponse, request.EndGetResponse, request) as HttpWebResponse;
            lblResult.Content = String.Format("The URL returned the following status code: {0}", response.StatusCode);
        }
        catch (Exception ex)
        {
            lblResult.Content = ex.Message;
        }
    }
    else
    {
        lblResult.Content = String.Empty;
    }
}

//private void ResponseCallback(IAsyncResult result)
//{
//    HttpWebRequest request = (HttpWebRequest)result.AsyncState;
//    HttpWebResponse response = (HttpWebResponse)request.EndGetResponse(result);
//    lblResult.Dispatcher.BeginInvoke(new Action(() =>
//    {
//        lblResult.Content = String.Format("The URL returned the following status code: {0}", response.StatusCode);
//    }));
//}
```

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

- Introducing **multithreading** into your applications has many advantages in terms of **performance and responsiveness**.
 - However, it 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.
- one can use the **lock** keyword to apply a **mutual-exclusion lock** to critical sections of code
 - A **mutual-exclusion lock** means: 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 }**
- For this:
 - Create a **private object** to apply the lock to
 - only one thread can access that object at any one time
 - 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.
}
```

```
public bool MakeCoffees(int coffeesRequired)
{
    // This condition will never be true unless you comment out the lock statement.
    if (stock < 0)
    {
        throw new Exception("Stock cannot be negative!");
    }

    lock (coffeeLock)
    {
        // Check that there is sufficient stock to fulfil the order.
        if (stock >= coffeesRequired)
        {
            // Introduce a delay to make thread contention more likely.
            Thread.Sleep(500);

            Console.WriteLine(String.Format("Stock level before making coffee: {0}", stock));
            Console.WriteLine("Making coffee...");
            stock = stock - coffeesRequired;
            Console.WriteLine(String.Format("Stock level after making coffee: {0}", stock));
            return true;
        }
        else
        {
            Console.WriteLine("Insufficient stock to make coffee");
            return false;
        }
    }
}
```


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**.
 - 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 **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 includes a set of collections that are designed for thread-safe access

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

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.

Module Review and Takeaways

- **Question:** 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?
 - () Option 1: task1.Wait();
task2.Wait();
task3.Wait();
 - () Option 2: Task.WaitAll(task1, task2, task3);
 - () Option 3: Task.WaitAny(task1, task2, task3);
 - () Option 4: Task.WhenAll(task1, task2, task3);
 - () Option 5: Task.WhenAny(task1, task2, task3);
- **Question:** 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?
 - () Option 1: public async IEnumerable<string> GetCoffees(string country, int strength)
 - () Option 2: public async Task<string> GetCoffees(string country, int strength)
 - () Option 3: public async Task<IEnumerable<string>> GetCoffees(string country, int strength)
 - () Option 4: public async Task GetCoffees(string country, int strength, out string result)
 - () Option 5: public async Task GetCoffees(string country, int strength, out IEnumerable<string> result)
- **Question:** You want to ensure that no more than five threads can access a resource at any one time. Which synchronization primitive should you use?
 - () Option 1: The ManualResetEventSlim class.
 - () Option 2: The SemaphoreSlim class.
 - () Option 3: The CountdownEvent class.
 - () Option 4: The ReaderWriterLockSlim class.
 - () Option 5: The Barrier class.