Lecture 13b

Asynchronous Programming with `async` and `await`

OBJECTIVES

In this lecture you will learn:

- What asynchronous programming is and how it can improve the performance of your apps.
- Use the **`async`** modifier to indicate that a method is asynchronous.
- Use an **`await`** expression to wait for an asynchronous task to complete execution so that an **`async`** method can continue its execution
- Take advantage of multicore processors by executing tasks asynchronously via features of the **Task Parallel Library** (TPL)
- Use **`Task`** method **`WhenAll`** to wait for multiple tasks to complete before an **`async`** method can continue its execution.
- Time multiple tasks running on single-core and dual-core
- systems (*with the same processor speeds*) to determine the performance improvement when these tasks are run on a dual-core system.
- Use a **`WebClient`** to invoke a web service asynchronously.

Outline

1 Introduction

2 Basics of async and await

3 Executing an Asynchronous Task from a GUI App

3.1 Asynchronous Execution of Two Compute Intensive Tasks

3.2 Awaiting Multiple Tasks

4 Problems to solve

1 Introduction

1 Introduction

The human body performs a great variety of operations *in **parallel-** or **concurrently.***

Computer tasks similarly can proceed independently of one another. Such tasks are said to execute asynchronously and are referred to as **asynchronous tasks**.

Only computers that have multiple processors or cores can *truly execute multiple asynchronous* tasks concurrently (**parallel task processing**). Visual C# apps can have multiple t**hreads of execution, where** each thread has its **own method-call stack**, allowing it to execute concurrently (**in parallel**) with other threads while **sharing with them application- wide resources** such as memory and processors. This capability is called **multithreading.**

1 Introduction

Operating systems on **single-core computers create the illusion** of concurrent execution by rapidly switching between activities (threads), but on such computers only a *single instruction can execute at once. In this case we speak about **concurrent task processing***.

Today's multi-core computers, smartphones and tablets enable computers to perform tasks truly concurrently. *In this case we speak about **parallel task processing***.

To take full advantage of multi-core architecture you need to write applications that can process tasks *asynchronously*. ***Asynchronous programming is a technique for writing apps*** containing tasks that can execute asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks.

1 Introduction

Visual C# 2012- 2013 introduces the **`async`** modifier and await operator to greatly simplify asynchronous programming, reduce errors and enable your apps to take advantage of the processing power in today's multicore computers, smartphones and tablets.

In .NET 4.5, many classes for **web access, file processing, networking, image processing** and more have been **updated with new methods** that return **`Task`** objects for use with **`async`** and **`await`**, so you can take advantage of this new asynchronous programming model.

This **lecture** presents a simple **introduction to asynchronous programming** with **`async`** and **`await`**.

Links to resources: MSDN, Async/Await FAQ, BEST PRACTICES FOR C# ASYNC/AWAIT

2 Basics of async and await

Before **`async`** and **`await`**, it was common for a method that was called *synchronously (i.e.*, performing tasks one after another in order) in the calling thread to **launch a long-running task** *asynchronously and to **provide** that task with **a callback method** (or, in some cases, register* an event handler) that would be invoked once the asynchronous task ***completed***.

This style of coding is simplified with **`async`** and **`await`**.

2 Basics of async and await

***`async`* Modifier**

The ***`async`*** **modifier indicates that a method or lambda expression** contains **at least one *`await`*** expression. An ***`async`*** method **executes its body in the same thread as the calling metho**d. (*Throughout the remainder of this lecture, we'll use the term "method" to mean "method or lambda expression.*")

2 Basics of async and await

***`await`* Expression**

An **`await` expression, which can appear *only in an* `async` *method, consists of the* `await` operator followed by an expression that returns an *awaitable entity- typically a* `Task` *object*** , though it is possible to create your own awaitable entities

(Asynchronous Programming with Async and Await)

2 Basics of async and await

When an `async` method encounters an `await` **expression:**

- **If the asynchronous task has already completed**, the `async` method simply **continues executing**.
- **Otherwise**, program control returns to the `async` method's caller **until** the asynchronous task **completes execution**. This **allows the caller to perform other work that does not depend on the results of the asynchronous task**.

When the asynchronous task completes, control returns to the `async` method and continues with the next statement after the `await` expression.

2 Basics of async and await

Note: ***`async`*, *`await`* and *`Threads`***

The **`async`** and **`await`** **mechanism does *not* *create new threads***. ***If any threads are required***, the **method that you call to start an asynchronous task** on which you **`await`** the results is responsible for **creating the threads** that are used to perform the asynchronous task.

`Async` methods **don't require multithreading** because an **`async`** method doesn't run on its own thread. **The method runs on the current synchronization context** and **uses time on the thread only when the method is active**.

2 Basics of async and await

Note: ***`async`*, *`await`* *and* *`Threads`***

You can use **`Task.Run`** to **move CPU-bound work to a background thread**, but a background thread doesn't help with a process that's just waiting for results to become available.

In the following examples (**Section 3**), we'll show how to use class **`Task`**'s **`Run`** method in several examples to **start new threads of execution** for executing tasks asynchronously. **`Task`** method **`Run`** returns a **`Task`** on which a method can **`await`** the result.

2 Basics of async and await

Note: ***async***, ***await*** *and* ***Threads***

The ***async- based*** approach to asynchronous programming is **preferable to existing approaches in almost every case**. In particular, this approach is **better** than ***BackgroundWorker*** for **IO-bound operations** because the **code is simpler and you don't have to guard against race conditions**. In **combination with *Task.Run***, ***async*** programming is **better** than ***BackgroundWorker*** for **CPU-bound** operations because ***async*** programming **separates the coordination details of running your code** from the work that ***Task.Run*** transfers to the ***Threadpool***.

2.1 Async Improves Responsiveness

Asynchrony is **essential for activities that are potentially blocking**, such as when your application accesses the web**. Access to a web resource** sometimes is **slow or delayed**.

If such an activity is blocked within a **synchronous process**, the **entire application must wait**.

In an **asynchronous process**, the **application can continue with other work that doesn't depend on the web resource** until the potentially blocking task finishes.

2.1 Async Improves Responsiveness

Asynchrony proves **especially valuable** for **applications that access the UI thread** because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use **asynchronous methods**, the application **continues to respond to the UI**. You can resize or minimize a window, for example, or you can close the application, if you don't want to wait for it to finish.

2.2 Async Methods Are Easier to Write

The **`async`** and **`await`** keywords in C# are the heart of **`async`** programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. **Asynchronous methods that you define** by using **`async`** and **`await`** are referred to as **`Async`** **methods**.

The **following example (`Sample1.zip`)** shows an **`Async`** method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony

```
// Three things to note in the signature:
//  - The method has an async modifier.
//  - The return type is Task or Task<T>. (See "Return Types" section.)
//    Here, it is Task<int> because the return statement returns an integer.
//  - The method name ends in "Async."
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    HttpClient client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    // The return statement specifies an integer result.
    // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    return urlContents.Length;
}
```

2.2 Async Methods Are Easier to Write

The **following characteristics summarize** what makes the previous example an **`Async`** method.

- The **method signature** includes an **`async`** **modifier**.
- **The name** of an **`Async`** method, by convention, ends with an "**`Async`**" **suffix**.
- The **return type** is one of the following types:
  - ✓**`Task<TResult>`** if your method has a **`return`** statement in which the **operand** has type **`TResult`**.
  - ✓**`Task`** if your method has **no return statement** or **has a return statement with no operand**.
  - ✓**`void`**, if you're writing an **`async`** event handler.

2.2 Async Methods Are Easier to Write

Note:

The .NET Framework 4.5 contains many members that work with `async` and `await`. You can recognize these members by the "`Async`" suffix that's attached to the member name and a return type of Task or `Task<TResult>`.

For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

2.2 Async Methods Are Easier to Write

– **`Task<T>`** referred to as **awaitable type**: If the calling method **is to receive a value of type T back** from the call, the return type of the **`Async`** method must be **`Task<T>`**. The calling method will then get the value of type **`T`** by reading the Task's **`Result`** **property**, as shown in the following code from a calling method:

```
Task<int> value = DoStuff.CalculateSumAsync( 5, 6 );
...
Console.WriteLine( "Value: {0}", value.Result );
```

– **`Task`** referred to as **awaitable type** : If the calling **method doesn't need a return value** from the **`Async`** method, but needs to be able to **check on the `Async` method's state**, then the **`Async`** method can return an object of type `Task`. In this case, **if there are any return statements** in the **`Async`** method, they must not return anything. The following code sample is from a calling method:

```
Task someTask = DoStuff.CalculateSumAsync(5, 6);
...
someTask.Wait();
```

2.2 Async Methods Are Easier to Write

```
 public async Task NewStuffAsync()
{
  // Use await
  await ...
}

public Task MyOldTaskParallelLibraryCode()
{
  // Note that this is not an async method, so we can't use await in here.
  ...
}

public async Task ComposeAsync()
{
  // We can await Tasks, regardless of where they come from.
  await NewStuffAsync();
  await MyOldTaskParallelLibraryCode();
}
```

One important point about awaitables is this: it **is the type that is awaitable**, **not the method returning the type**. In other words, you can **await the result of an async method** that returns **`Task`** ... because the method returns **`Task`**, not because it's **`async`**. So you can also await the result of a non-async method that returns **`Task`**

2.2 Async Methods Are Easier to Write

Tip:

If you have a very simple asynchronous method, you may be able to write it without using the **`await`** keyword (e.g., using **`Task.FromResult`**). If you **can write it without** **`await`**, then you **should** write it without **`await`**, and remove the **`async`** keyword from the method. A **non-async method** returning **`Task.FromResult`** is **more efficient** than an **`async`** method returning a value

```
public  Task<int> DoWork() {
    Thread.Sleep(220);
    return Task.FromResult(9999999);
}
// better than
public async Task<int> DoWork() {
     await Task.Delay(220);
     return 9999999;
}
```

2.2 Async Methods Are Easier to Write

```
// TASK<T> EXAMPLE
async Task<int> TaskOfT_MethodAsync()
{
    // The body of the method is expected to contain an awaited asynchronous
    // call.
    // Task.FromResult is a placeholder for actual work that returns a string.
    var today = await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

    // The method then can process the result in some way.
    int leisureHours;
    if (today.First() == 'S')
        leisureHours = 16;
    else
        leisureHours = 5;

    // Because the return statement specifies an operand of type int, the
    // method must have a return type of Task<int>.
    return leisureHours;
}
```

```
// Call and await the Task<T>-returning async method in the same statement.
int result1 = await TaskOfT_MethodAsync();
```

2.2 Async Methods Are Easier to Write

– **`void:`** If the calling method just wants the `Async` method to execute, but **doesn't need any further interaction** with it (this is sometimes called fire and forget), the **`Async`** method can have a return type of **`void`**. In this case, as with the previous case, if there **are any return statements** in the **`Async`** method, they must **not return anything**.

Async `void` Method

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

2.2 Async Methods Are Easier to Write

- An **`Async`** method can **`have any number of formal parameters`** of **`any types`**. **`None`** of the parameters, however, can be **`out`** or **`ref`** parameters.
- Besides methods, lambda expressions and anonymous methods can also act as **`async`** objects.

Async Lambda expression

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

2.2 Async Methods Are Easier to Write

Note:

You can **ignore the convention** to use the **`Async`** suffix where an event, base class, or interface contract suggests a different method name. For example, you shouldn't rename common event handlers, such as **`Button1_Click`**

Note:

The **`Async`** method **usually** **includes** at least one **`await`** expression, which **marks a point** where the **method can't continue until the awaited asynchronous operation is complete**. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

2.2 Async Methods Are Easier to Write

If you specify that a method is an async method by using an **`async`** modifier, you enable the following **two capabilities**.

- The marked **`async`** method can use **`await`** to **designate suspension points**. The **`await`** operator tells the compiler that the **`async`** method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.
- The suspension of an **`async`** method at an **`await`** expression doesn't constitute an exit from the method, and **finally** blocks don't run.
- The marked **`async`** method can itself be awaited by methods that call it.

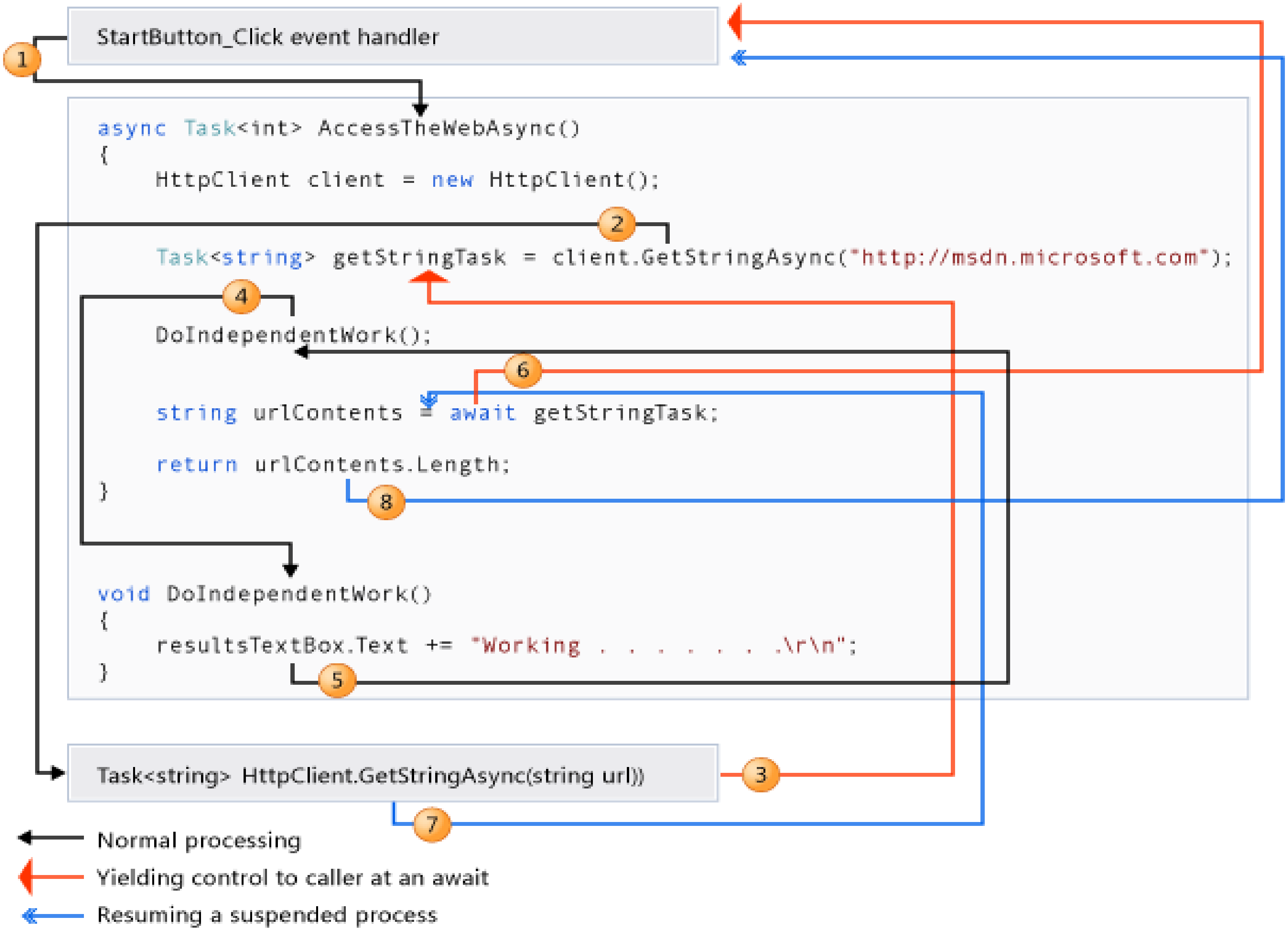
2.3 Async Method control flow

The **most important thing to understand** in asynchronous programming is **how the control flow moves from method to method**.

The following diagram leads you through the process.

Detailed explanation of each step is provided in
MSDN: Control Flow in Async Programs

and
It's All About the SynchronizationContext


StartButton_Click event handler
async Task<int> AccessTheWebAsync()
{
HttpClient client = new HttpClient();
Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
DoIndependentWork();
string urlContents = await getStringTask;
return urlContents.Length;
}
void DoIndependentWork()
{
resultsTextBox.Text += "Working . . . .\r\n";
}
Task<string> HttpClient.GetStringAsync(string url))
1
2
3
4
5
6
7
8
Normal processing
Yielding control to caller at an await
Resuming a suspended process

2.3 Async Method control flow

The numbers in the diagram correspond to the following **steps**.

1. An **event handler calls** and awaits the **`AccessTheWebAsync`** async method.
2. **`AccessTheWebAsync`** creates an **`HttpClient`** instance and calls the **`GetStringAsync`** asynchronous method to **download the contents of a website** as a **`string`**.
3. Something happens in **`GetStringAsync`** that suspends its progress. **Perhaps it must wait** for a website to download or some other blocking activity. **To avoid blocking** resources, **`GetStringAsync` yields control** to its caller, **`AccessTheWebAsync`**. **`GetStringAsync`** returns a **`Task<TResult>`** where **`TResult`** is a **`string`**, and **`AccessTheWebAsync`** assigns the task to the **`getStringTask`** variable. The **task represents the ongoing process** for the call to **`GetStringAsync`**, with **a commitment to produce an actual string** value **when the work is complete**

2.3 Async Method control flow

4. Because **`getStringTask`** hasn't been awaited yet, **`AccessTheWebAsync`** **can continue** with other **work that doesn't depend on the final result** from **`GetStringAsync`**. That work is represented by a **call to the synchronous** method **`DoIndependentWork`**.

5. **`DoIndependentWork`** is a **synchronous method that does its work and returns to its caller**.

6. **`AccessTheWebAsync`** has **run out of work that it can do without a result from** **`getStringTask`**. **`AccessTheWebAsync`** next wants to calculate and **return the length** of the downloaded **`string`**, but the method can't calculate that value until the method has the string.
   Therefore, **`AccessTheWebAsync`** **uses** an **`await`** operator to **suspend its progress** and to yield control to the method that called **`AccessTheWebAsync`**. **`AccessTheWebAsync`** returns a **`Task<int>`** to the caller. The **task represents a promise to produce an integer result** that's the length of the downloaded string.

2.3 Async Method control flow

Note:

If **`GetStringAsync`** (and therefore **`getStringTask`**) is complete **before** **`AccessTheWebAsync`** awaits it, **control remains in** **`AccessTheWebAsync`**.

The expense of suspending and then returning to **`AccessTheWebAsync`** **would be wasted, if** the called asynchronous process (**`getStringTask`**) **has already completed** and **`AccessTheWebSync`** doesn't have to wait for the final result.

2.3 Async Method control flow

Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from **`AccessTheWebAsync`** before awaiting that result, or the caller might await immediately. The event handler is waiting for **`AccessTheWebAsync`**, and **`AccessTheWebAsync`** is waiting for **`GetStringAsync`**.

7. **`GetStringAsync`** completes and produces a string result. The **`string`** result isn't returned by the call to **`GetStringAsync`** in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the **`string`** result is **stored in the task** that represents the completion of the method, **`getStringTask`**. The **`await`** **operator retrieves the result** from **`getStringTask`**. The assignment statement assigns the retrieved result to **`urlContents`**.
8. When **`AccessTheWebAsync`** has the **`string`** result, the method can calculate the **`length`** of the string. Then the work of **`AccessTheWebAsync`** is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the **`length`** result.

2.3 Async Method control flow

The **structure of the body** of an **`Async`** method has **three distinct regions**. The three regions are the following:

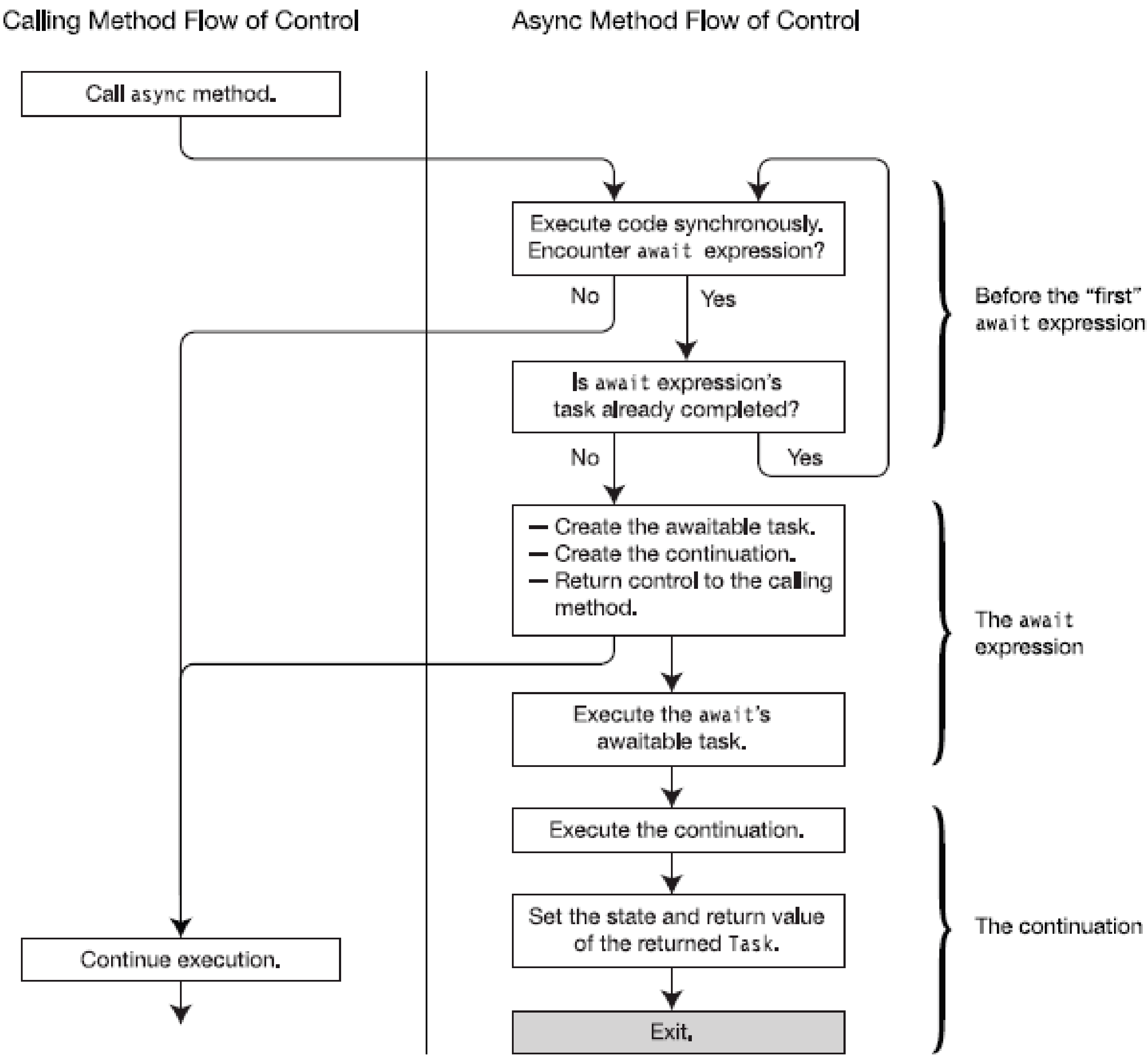
- ***Before*** *the first* ***`await`*** *expression: This includes all the code at the beginning of the* method up until the first ***`await`*** expression. This region should only **contain a small amount of code** that doesn't require too much processing.
- *The* ***`await`*** *expression: This expression* ***represents the task to be performed*** asynchronously.
- *The* ***continuation****: This is the rest of the code in the method, following the* ***`await`*** expression. This is packaged up along with its execution environment, which includes the information about which thread it's on, the values of the variables currently in scope, and other things it'll need in order to **resume execution later**, after the ***`await`*** expression completes

2.3 Async Method control flow

The **flow of control starts** with the code **before** the first **`await`** expression, and executes normally (synchronously) until it encounters the first **`await`**. This region actually ends at the first **`await`** expression, where the **`await's`** task has not already completed (which should be the vast majority of the time). If the **`await's`** task has already completed, the method continues executing synchronously. The process is repeated if another **`await`** is encountered.

When the **`await`** expression is reached, the **`async`** method returns control to the calling method. If the method's **`return`** type is of type **`Task`** or **`Task<T>`**, the method creates a **`Task`** object that represents both the task to be done asynchronously and the continuation, and returns that **`Task`** to the calling method

The Flow of Control in an `Async` Method



2.3 Async Method control flow

One thing that people are sometimes confused about is the type of the object returned when the first **await** in the **Async** method is encountered. The **type returned is the type listed as** the **return** type in the header of the **async** method; **it has nothing to do with the type of the value returned by** the **await** expression.

In the code below, for example, the **await** expression returns a **string**. But during execution of the method, when that **await** expression is reached, the **Async** method returns to the calling method an object of **Task<int>** because **that's the return type of the method**

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;
    return urlContents.Length;
}
```

2.4 Build asynchronous methods

Consider **the difference** between **synchronous** and **asynchronous** behavior. A synchronous method returns when its work is complete (step 5), but an **`Async`** method returns a task value **when its work is suspended** (steps 3 and 6). When the **`Async`** method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task

For **illustration and comparison**, we'll start by looking at an example (**`Sample2`**) that does ***not use asynchrony***, *and* then **compare** it to a similar program (**`Sample3`**) **that uses asynchrony**.

2.4 Build asynchronous methods

In the code **Sample2** shown below, method DoRun is a method of class **`MyDownloadString`** that does the following:

- It **creates and starts an object** of class **`Stopwatch`**, which is in the **`System.Diagnostics`** namespace. It uses this **`Stopwatch`** timer to time the various tasks performed in the code.
- It then **makes two calls** to method **`CountCharacters`**, which **downloads** the content of the web site and **returns** the **`number`** of characters the web site contains. The web site is specified as a **`URL`** string given as the second parameter. Similarly to **Sample1**, where we have use **`class HttpClient (with only Async methods, method GetStringAsync())`**, here we use **`class WebClient (methods DownloadString() and DownloadStringAsync())`**
- It then **makes four calls** to method **`CountToALargeNumber`**. This method is **just make-work** that represents **a task that takes a certain amount of time**. It just loops the given number of times.
- Finally, it **prints out the number of characters** that were found for the two web sites.

2.4 Build asynchronous methods

```
class MyDownloadString // Sample 2
{
    Stopwatch sw = new Stopwatch();
    public void DoRun()
    {
        const int LargeNumber = 6000000;
        sw.Start();
        int t1 = CountCharacters( 1, "http://www.microsoft.com" );
        int t2 = CountCharacters( 2, "http://www.cs.com" );
        CountToALargeNumber( 1, LargeNumber );
        CountToALargeNumber( 2, LargeNumber );
        CountToALargeNumber( 3, LargeNumber );
        CountToALargeNumber( 4, LargeNumber );
        Console.WriteLine( "Chars in http://www.microsoft.com : {0}", t1 );
        Console.WriteLine( "Chars in http://www.cs.com: " +   {0}", t2 );
    }
```

2.4 Build asynchronous methods

```
 private int CountCharacters( int id, string uriString )
    {
        WebClient wc1 = new WebClient();
        Console.WriteLine( "Starting call {0} : {1, 4:N0} ms", id,
                                              sw.Elapsed.TotalMilliseconds );
        string result = wc1.DownloadString( new Uri( uriString ) );
        Console.WriteLine( " Call {0} completed: {1, 4:N0} ms", id,
                                              sw.Elapsed.TotalMilliseconds );
        return result.Length;
    }
    private void CountToALargeNumber( int id, int value )
    {
        for ( long i=0; i < value; i++ )    ;
        Console.WriteLine( " End counting {0} : {1, 4:N0} ms", id,
                                                sw.Elapsed.TotalMilliseconds );
    }
}
```

2.4 Build asynchronous methods

```
class Program
{
    static void Main()
    {
        MyDownloadString ds = new MyDownloadString();
        ds.DoRun();
    }
}
```

```
Starting call 1 :                        1 ms
Call 1 completed:                        178 ms
Starting call 2 :                        178 ms
Call 2 completed:                        504 ms
End counting 1 :                         523 ms
End counting 2 :                         542 ms
End counting 3 :                         561 ms
End counting 4 :                         579 ms
Chars in http://www.microsoft.com :      1020
Chars in http://www.cs.com:              4699
```

2.4 Build asynchronous methods

C#'s new **`async/await`** feature allows us to improve the performance of the program. The code, rewritten to use this feature, is shown below. Notice the following:

- When method **`DoRun`** calls **`CountCharactersAsync`**, **`CountCharactersAsync`** **returns almost immediately, and before** it actually does the work of downloading the characters. It returns to the calling method a placeholder object of type **`Task<int>`** that **represents the work it plans to do**, which will eventually "**`return`**" an **`int`**.
- This allows method **`DoRun`** to continue on its **way without having to wait for the actual work to be done**. Its next statement is another call to **`CountCharactersAsync`**, which does the same thing, returning another **`Task<int>`** object.

2.4 Build asynchronous methods

- **DoRun** can then **continue on and make the four calls** to **CountToALargeNumber**, while the two calls to **CountCharactersAsync** continue to do their work, which consists mostly of waiting.
- The **last two lines** of method **DoRun** **retrieve** the results from the **Tasks** returned by the **CountCharactersAsync** calls making use of the **Result** property of class **Task**. If a result isn't ready yet, **execution blocks and waits until it is.**

2.4 Build asynchronous methods

```
using System.Threading.Tasks;
class MyDownloadString // Sample 3
{
   Stopwatch sw = new Stopwatch();
   public void DoRun() // not marked as Async method, has no await keyword
   {
     const int LargeNumber = 6000000;
     sw.Start(); // Start Stopwatch
     Task<int> t1 = CountCharactersAsync( 1,"http://www.microsoft.com" );
     Task<int> t2 = CountCharactersAsync( 2,"http://www.cs.com" );
     CountToALargeNumber( 1, LargeNumber );
     CountToALargeNumber( 2, LargeNumber );
     CountToALargeNumber( 3, LargeNumber );
     CountToALargeNumber( 4, LargeNumber );
     Console.WriteLine( "Chars in http://www.microsoft.com: {0}",
                                                              t1.Result );
      Console.WriteLine( "Chars in http://www.cs.com: {0}", t2.Result );
   }
```

Retrieve the results in a completed `Task <int>`.

2.4 Build asynchronous methods

```
   private async Task<int> CountCharactersAsync( int id, string site )
   { // async  is referred to as “Contextual keyword”
      WebClient wc = new WebClient();
      Console.WriteLine( "Starting call {0}: {1, 4:N0} ms",
                                         id, sw.Elapsed.TotalMilliseconds );
      string result = await wc.DownloadStringTaskAsync(new Uri( site ) );
     // await is referred to as “Contextual keyword”
      Console.WriteLine( " Call {0} completed : {1, 4:N0} ms",
                                         id, sw.Elapsed.TotalMilliseconds );
      return result.Length; // returns int
   }
   private void CountToALargeNumber( int id, int value )
   {
      for ( long i=0; i < value; i++ )
         ;
      Console.WriteLine( " End counting {0} : {1, 4:N0} ms",
                                      id, sw.Elapsed.TotalMilliseconds );
   }
}
```

2.4 Build asynchronous methods

```
class Program
{
    static void Main()
    {
        MyDownloadString ds = new MyDownloadString();
        ds.DoRun();
    }
}
```

```
Starting call 1 :                         12 ms
Starting call 2 :                         60 ms
End counting 1 :                          80 ms
End counting 2 :                          99 ms
End counting 3 :                          118 ms
Call 1 completed:                         124 ms
End counting 4 :                          138 ms
Chars in http://www.microsoft.com :       1020
Call 2 completed:                         387 ms
Chars in http://www.cs.com:               4699
```

Sample 3 execution with **Async methods** (**Total 1018** ms)

```
Starting call 1 :                         1 ms
Call 1 completed:                         178 ms
Starting call 2 :                         178 ms
Call 2 completed:                         504 ms
End counting 1 :                          523 ms
End counting 2 :                          542 ms
End counting 3 :                          561 ms
End counting 4 :                          579 ms
Chars in http://www.microsoft.com :       1020
Chars in http://www.cs.com:               4699
```

Sample 2 synchronous execution

(Total 3066 ms)

2.4 Build asynchronous methods

The new version is **32%** **faster** than the previous version. It gains this time by performing the four calls to **`CountToALargeNumber`** during the time it's waiting for the responses from the web sites in the two **`CountCharactersAsync`** method calls.

All this was **done on the main thread**, we did **not create any additional threads!**

2.4.1 Async Lambdas

You can easily create lambda expressions and statements that **incorporate asynchronous processing** by using the **`async`** and **`await`** keywords.

For example, the following **`Windows Forms`** example contains an event handler that calls and **`awaits`** an **`async`** method,

`ExampleMethodAsync`

2.4.1 Async Lambdas

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nReturn to Click event handler.\r\n";
    }
    async Task ExampleMethodAsync()
    {
        // Simulate a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

2.4.1 Async Lambdas

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nReturn to Click event handler.\r\n";
        };
    }
    async Task ExampleMethodAsync()
    {
        // Simulate a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Event handler by using an async lambda

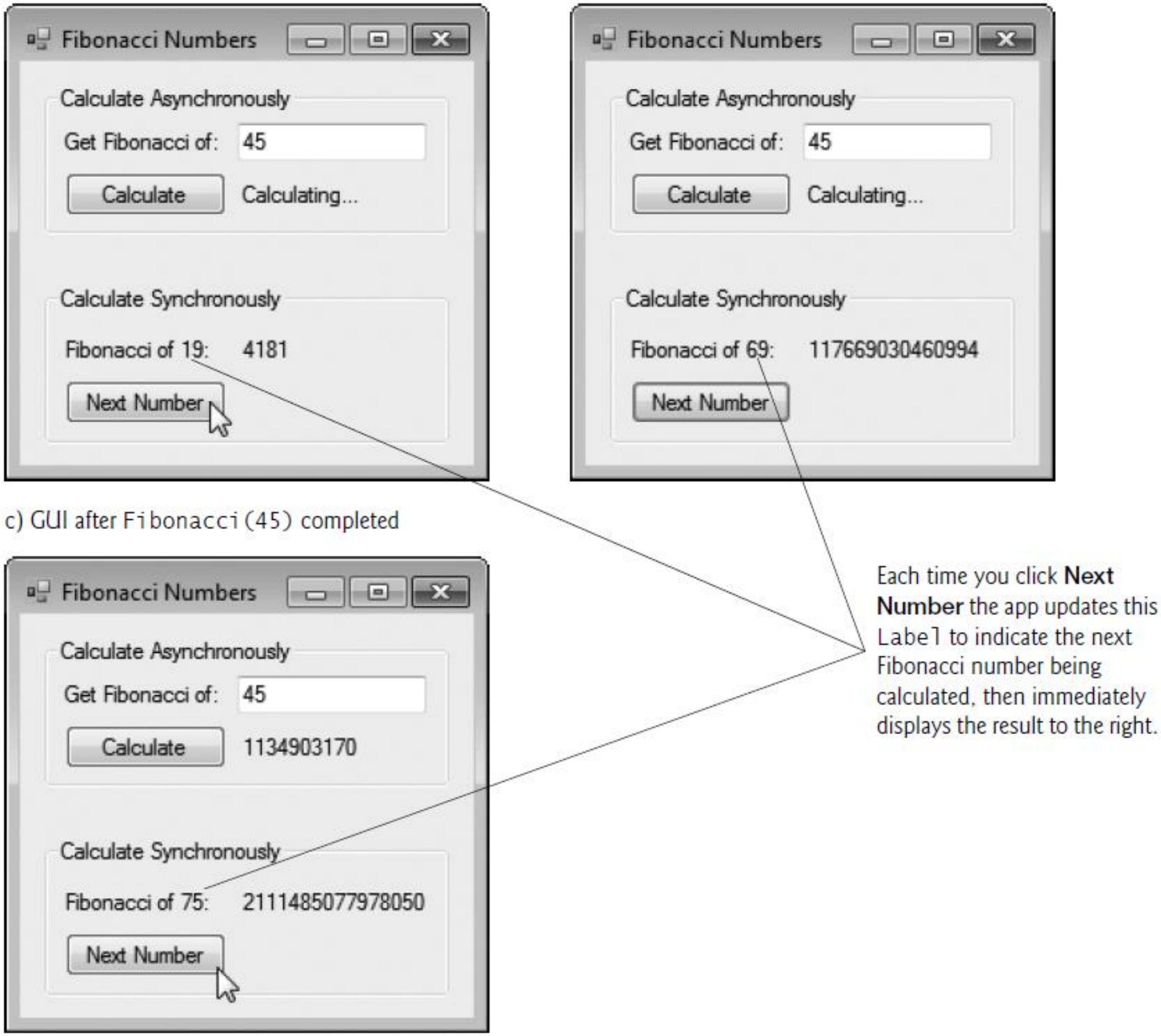
3.1Executing an Asynchronous Task from a GUI App

Consider a **CPU-bound work, for example, *Calculating Fibonacci Numbers Recursively*.** The Fibonacci series can be defined *recursively as follows:*

$$\text{Fibonacci}(0) = 0$$
$$\text{Fibonacci}(1) = 1$$
$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

This rapidly gets out of hand as *n gets larger. Calculating* only the 20th Fibonacci number would require on the order of 220 or about a million calls, calculating the 30th Fibonacci number would require on the order of 230 or about a billion calls, and so on. If this calculation were to be performed ***synchronously***, *the GUI would freeze for that amount of time and the* user would not be able to interact with the app (as we'll demonstrate in Fig. 28.2). We launch the calculation ***asynchronously and have it execute on a separate thread so the GUI*** remains ***responsive***.

3.1Executing an Asynchronous Task from a GUI App



3.1Executing an Asynchronous Task from a GUI App

```
// Fig. 28.1: FibonacciForm.cs
// Performing a compute-intensive calculation from a GUI app
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace FibonacciTest
{
   public partial class FibonacciForm : Form
   {
      private long n1 = 0; // initialize with first Fibonacci number
      private long n2 = 1; // initialize with second Fibonacci number
      private int count = 1; // current Fibonacci number to display

      public FibonacciForm()
      {
         InitializeComponent();
      } // end constructor

      // start an async Task to calculate specified Fibonacci number
      private async void calculateButton_Click(
         object sender, EventArgs e )
      {
```

Fig. 5.1 | Performing a compute-intensive calculation from a GUI app. (Part 1 of 6.)

3.1Executing an Asynchronous Task from a GUI App

```
            // retrieve user's input as an integer
            int number = Convert.ToInt32( inputTextBox.Text );

            asyncResultLabel.Text = "Calculating...";

            // Task to perform Fibonacci calculation in separate thread
            Task< long > fibonacciTask =
               Task.Run( () => Fibonacci( number ) );

            // wait for Task in separate thread to complete
            await fibonacciTask;

            // display result after Task in separate thread completes
            asyncResultLabel.Text = fibonacciTask.Result.ToString();
         } // end method calculateButton_Click

```

Fig. 5.1 | Performing a compute-intensive calculation from a GUI app. (Part 2 of 6.)

Initiates the call to method Fibonacci **in a separate thread and displays the result**

3.1Executing an Asynchronous Task from a GUI App

The **Calculate button's event handler (lines 21–38) initiates the call to method Fibonacci** in a separate thread and displays the results when the call completes. The method is declared **`async`** (line 21) to indicate to the compiler that the method will initiate an asynchronous task and **`await`** the results. In effect, an **`async`** method allows you to write code that looks like it executes sequentially, while the compiler deals with the complicated issues of managing asynchronous execution. This makes your code easier to write, modify and maintain, and reduces errors

3.1Executing an Asynchronous Task from a GUI App

Lines 30–31 create and start a **Task (namespace System.Threading.Tasks). A Task promises** to return a result *at some point in the future. Class* **Task** *is part of .NET's Task Parallel Library (TPL) for asynchronous programming. The version of* **class Task***'s* ***static*** *method* **Run used in line 31 receives a Func<TResult> delegate** as an argument and executes a method in a *separate thread. The delegate* **Func<TResult>** represents any method that takes ***no arguments and returns a result***, ***so the*** name of any method that takes no arguments and returns a result can be passed to **Run**. However, Fibonacci requires an argument, so line 31 use the *lambda expression*

```
() => Fibonacci( number )
```

which **takes *no arguments to encapsulate the call to Fibonacci with the argument number***

3.1Executing an Asynchronous Task from a GUI App

The lambda expression ***implicitly returns the result of the Fibonacci call*** *(a* **long***)*, *so it* meets the **Func<TResult>** delegate's requirements. In this example, **Task's** static method **Run** creates and returns a **Task<long>** that represents the task being performed in a separate thread. The compiler *infers the type long from the return type of method Fibonacci.*

Next, **line 34 awaits the result** of the **fibonacciTask** that's executing asynchronously. If the **fibonacciTask** is already complete, execution continues with line 37. Otherwise, control returns to **calculateButton_Click's** caller (the GUI event handling thread) until the result of the **fibonacciTask** is available. This allows the GUI to remain *responsive while* the Task executes. Once the **Task** completes, **calculateButton_Click** continues execution at line 37, which uses **Task** property **Result to get the value returned by Fibonacci** and display it on **asyncResultLabel**.

3.1Executing an Asynchronous Task from a GUI App

```
        // calculate next Fibonacci number iteratively
        private void nextNumberButton_Click( object sender, EventArgs e )
        {
           // calculate the next Fibonacci number
           long temp = n1 + n2; // calculate next Fibonacci number
           n1 = n2; // store prior Fibonacci number in n1
           n2 = temp; // store new Fibonacci
           ++count;

           // display the next Fibonacci number
           displayLabel.Text = string.Format( "Fibonacci of {0}:", count );
           syncResultLabel.Text = n2.ToString();
        } // end method nextNumberButton_Click

```

Fig. 5.1 | Performing a compute-intensive calculation from a GUI app. (Part 3 of 6.)

3.1Executing an Asynchronous Task from a GUI App

```
        // recursive method Fibonacci; calculates nth Fibonacci number
        public long Fibonacci( long n )
        {
           if ( n == 0 || n == 1 )
              return n;
           else
              return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
        } // end method Fibonacci
     } // end class FibonacciForm
  } // end namespace FibonacciTest
```

Fig. 5.1 | Performing a compute-intensive calculation from a GUI app. (Part 4 of 6.)

3.1Executing an Asynchronous Task from a GUI App

It's important to note that an **`async`** method can perform other statements between those that launch an asynchronous **`Task`** and await the **`Task`**'s results. In such a case, the method continues executing those statements after launching the asynchronous Task until it reaches the await expression.

Lines 30–34 can be written more concisely as

```
long result = await Task.Run( () => Fibonacci( number ) );
```

In this case, the **`await`** operator unwraps and returns the **`Task's`** result- the long returned by method **`Fibonacci`**. You can then use the **`long`** value directly without accessing the **`Task's Result`** property

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

When you run any program, your program's **tasks compete for processor time** with the operating system, other programs and other activities that the operating system is running on your behalf. When you execute the next example, the time to perform the Fibonacci calculations can vary based on your computer's processor speed, number of cores and what else is running on your computer. It's like a drive to the university- the time it takes can vary based on traffic conditions, weather, timing of traffic lights and other factors.

Figure 28.3 also uses the recursive Fibonacci method, but **the two initial calls to Fibonacci execute in *separate threads***.

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
// Fig. 28.3: AsynchronousTestForm.cs
// Fibonacci calculations performed in separate threads
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace FibonacciAsynchronous
{
   public partial class AsynchronousTestForm : Form
   {
      public AsynchronousTestForm()
      {
         InitializeComponent();
      } // end constructor

      // start asynchronous calls to Fibonacci
      private async void startButton_Click( object sender, EventArgs e )
      {
         outputTextBox.Text =
            "Starting Task to calculate Fibonacci(46)\r\n";

         // create Task to perform Fibonacci(46) calculation in a thread
         Task< TimeData > task1 =
            Task.Run( () => StartFibonacci( 46 ) );
```

Initiates a first call to method Fibonacci **in a separate thread**

Fig. 5.3 | Fibonacci calculations performed in separate threads. (Part 1 of 6.)

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

Initiates a second call to method Fibonacci **in a separate thread**

```
         outputTextBox.AppendText(
            "Starting Task to calculate Fibonacci(45)\r\n" );

         // create Task to perform Fibonacci(45) calculation in a thread
         Task< TimeData > task2 =
            Task.Run( () => StartFibonacci( 45 ) );

         await Task.WhenAll( task1, task2 ); // wait for both to complete

         // determine time that first thread started
         DateTime startTime =
            ( task1.Result.StartTime < task2.Result.StartTime ) ?
            task1.Result.StartTime : task2.Result.StartTime;

         // determine time that last thread ended
         DateTime endTime =
            ( task1.Result.EndTime > task2.Result.EndTime ) ?
            task1.Result.EndTime : task2.Result.EndTime;
```

Fig. 5.3 | Fibonacci calculations performed in separate threads. (Part 2 of 6.)

3.3 Awaiting Multiple Tasks with Task Method WhenAll

In method **startButton_Click**, lines 23–24 and 30–31 use **Task** method **Run** to create and start **Tasks** that execute method **StartFibonacci** (lines 53–71)—one to calculate **Fibonacci(46)** and one to calculate **Fibonacci(45)**. To show the total calculation time, the app must wait for *both* **Task**s to complete *before* executing lines 36–49.

You can **wait for *multiple* Tasks to complete** by awaiting the result of **Task static** method **WhenAll** (line 33), which returns a **Task** that waits for *all* of **WhenAll's** argument **Tasks** to complete and places *all* the results in an array. In this app, the **Task**'s **Result** is a **TimeData[]**, because **both of WhenAll's** argument **Task**s **execute methods** that **return TimeData** objects. **This array can be used to iterate** through the results of the awaited **Task**s.

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
      // display total time for calculations
      outputTextBox.AppendText( String.Format(
         "Total calculation time = {0:F6} minutes\r\n",
         endTime.Subtract( startTime ).TotalMilliseconds /
         60000.0 ) );
   } // end method startButton_Click

   // starts a call to fibonacci and captures start/end times
   TimeData StartFibonacci( int n )
   {
      // create a TimeData object to store start/end times
      TimeData result = new TimeData();

      AppendText( String.Format( "Calculating Fibonacci({0})", n ) );
      result.StartTime = DateTime.Now;
      long fibonacciValue = Fibonacci( n );
      result.EndTime = DateTime.Now;

      AppendText( String.Format( "Fibonacci({0}) = {1}",
         n, fibonacciValue ) );
      AppendText( String.Format(
         "Calculation time = {0:F6} minutes\r\n",
         result.EndTime.Subtract(
            result.StartTime ).TotalMilliseconds / 60000.0 ) );
```

Fig. 5.3 | Fibonacci calculations performed in separate threads. (Part 3 of 6.)

3.4 Modifying a GUI from a Separate Thread

Method **`StartFibonacci`** (lines 53–71) specifies the task to perform- in this case, to call **`Fibonacci`** (line 60) to **perform the recursive calculation**, to **time the calculation** (lines 59 and 61), to **display the calculation's result** (lines 63–64) and **to display the time the calculation took** (lines 65–68).

The method returns a **`TimeData`** object (defined in this project's **`TimeData.cs`** file) that **contains the time before and after** each thread's call to **`Fibonacci`**.

Class **`TimeData`** contains public auto-implemented properties **`StartTime`** and **`EndTime`**, which we use in our timing calculations.

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
            return result;
        } // end method StartFibonacci

        // Recursively calculates Fibonacci numbers
        public long Fibonacci( long n )
        {
            if ( n == 0 || n == 1 )
                return n;
            else
                return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
        } // end method Fibonacci

        // append text to outputTextBox in UI thread
        public void AppendText( String text )
        {
            if ( InvokeRequired ) // not GUI thread, so add to GUI thread
                Invoke( new MethodInvoker( () => AppendText( text ) ) );
            else // GUI thread so append text
                outputTextBox.AppendText( text + "\r\n" );
        } // end method AppendText
    } // end class AsynchronousTestForm
} // end namespace FibonacciAsynchronous
```

Fig. 5.3 | Fibonacci calculations performed in separate threads. (Part 4 of 6.)

3.4 Modifying a GUI from a Separate Thread

Lines 58, 63 and 65 in **`StartFibonacci`** call our **`AppendText`** method (lines 83–89) to append text to the **`outputTextBox`**. GUI controls are designed to be manipulated *only* by the GUI thread—modifying a control from a non-GUI thread can corrupt the GUI, making it unreadable or unusable. **When updating a control from a non-GUI thread, you *must schedule* that update to be performed by the GUI thread**.

To do so in Windows Forms, you check the **`InvokeRequired`** **property** of **`class Form`** (line 85). If this property's value is **`true`**, the **code is executing in a non-GUI thread** and ***must not* update the GUI directly**. In this case, you call the **`Invoke`** method of **`class Form`** (line 86), which receives as an argument a **`Delegate`** representing the update to perform in the **`GUI`** thread.

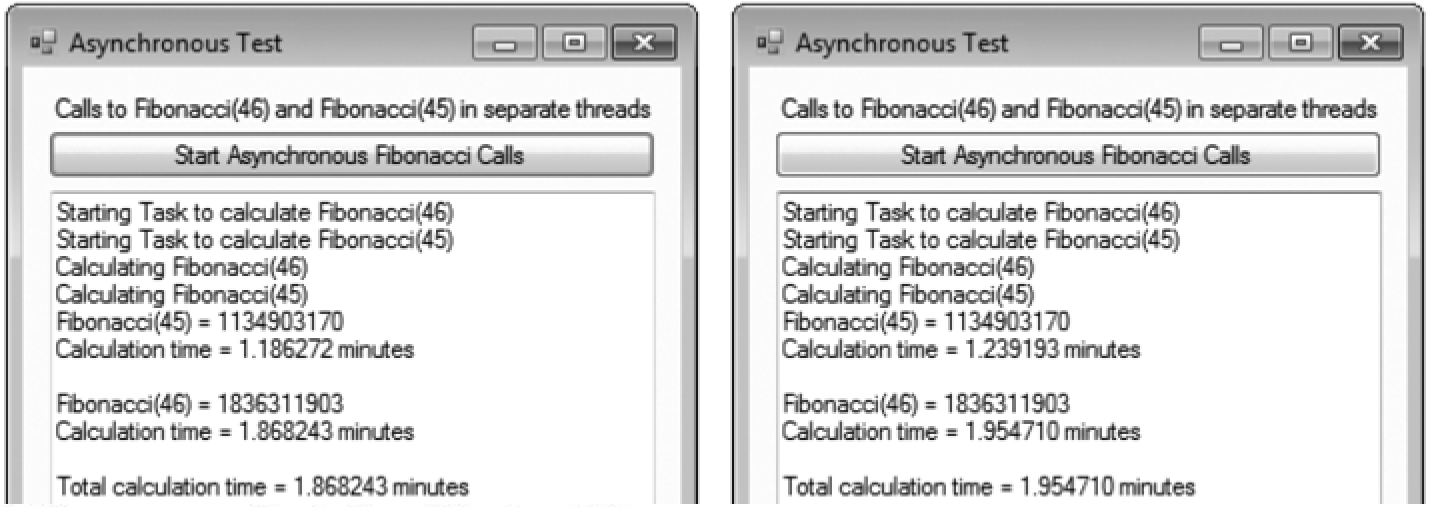
3.4 Modifying a GUI from a Separate Thread

In this example, we pass a **`MethodInvoker`** (namespace **`System.Windows.Forms`**), which is a **`Delegate`** that invokes a method with no arguments and a void return type. The **`MethodInvoker`** is initialized here with a *lambda expression* that calls **`AppendText`**. Line 86 *schedules* this **`MethodInvoker`** (delegate) to **execute in the GUI thread**. When that occurs, line 88 updates the **`outputTextBox`**.

(Similar concepts also apply to GUIs created with WPF and Windows 8 UI.)

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

a) Outputs on a Dual Core Windows 7 Computer



b) Outputs on a Single Core Windows 7 Computer

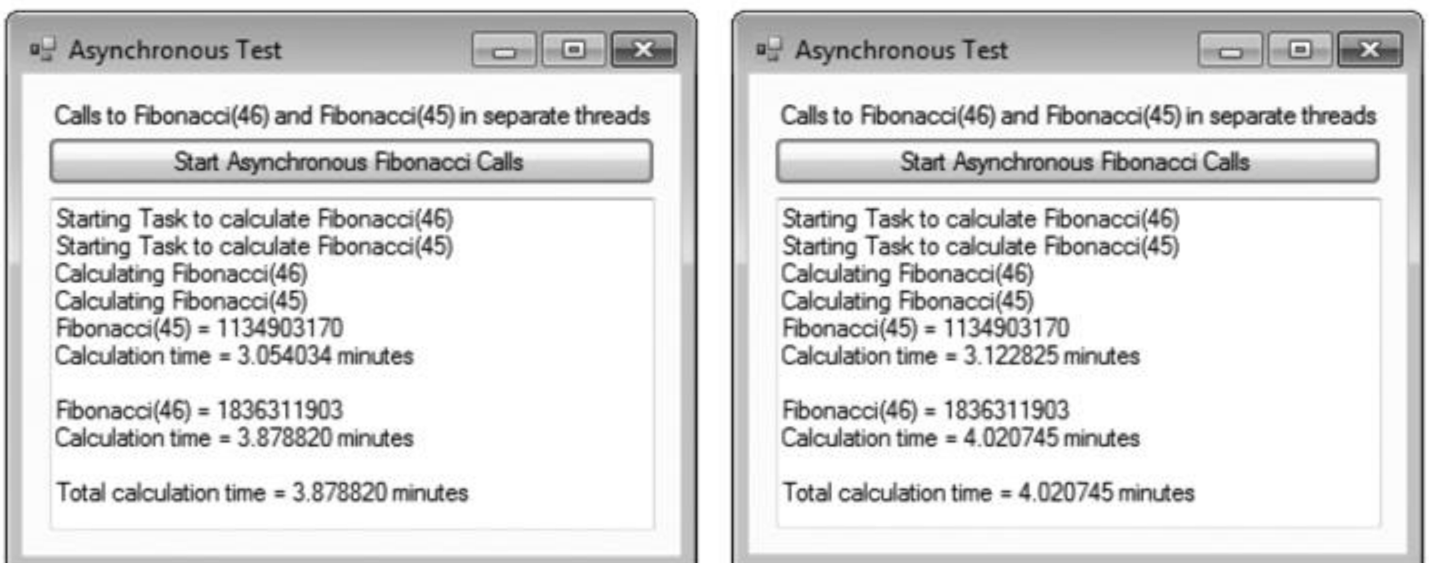


Fig. 5.3 | Fibonacci calculations performed in separate threads. (Part 6 of 6.)

3.2 Asynchronous Execution of Two Compute– Intensive Tasks

The **first two outputs** show the results on a ***dual-core*** **computer**. Though execution times varied, **the total time** to perform both Fibonacci calculations (in our tests) was **typically *significantly less* than the total time of the sequential execution**.

The last two outputs show that **executing calculations in multiple threads on a single core processor** can actually **take *longer* than simply performing them synchronously**, due to the overhead of sharing *one* processor among the app's threads, **all the other apps executing on the computer at the same time and the chores the operating system was performing**.

3.5 awaiting One of Several Tasks with Task Method WhenAny

Similar to **`WhenAll`**, class Task also provides static method **`WhenAny`**, which enables you **to wait for any one of several Tasks specified as arguments to complete**. **`WhenAny`** returns the **`Task`** that completes first. One **use** of **`WhenAny`** might be to **initiate several `Task`s** that perform **the same complex calculation on computers around the Internet**, then **wait for any one of those computers to send results back**. This would allow you to take advantage of computing power that's available to you to get the result as fast as possible. In this case, it's up to you to decide whether to cancel the remaining **`Task`s** or allow them to continue executing.

Another use of **`WhenAny`** might be to download several large files- one per **`Task`**. Eventually, you would like to immediately start processing the results from the first **`Task`** that returns. Next, make a new call to **`WhenAny`** for the remaining Tasks.

Problems to solve

Solve the problems in Lab14b.pdf