Lecture 14a

Multithreading

OBJECTIVES

In this lecture you will learn:

- What threads are and why they are useful.
- How threads enable you to manage concurrent activities.
- The life cycle of a thread.
- Thread priorities and scheduling.
- To create and execute `Threads`.
- Thread synchronization.
- What producer/consumer relationships are and how they are implemented with multithreading.
- To display output from multiple threads in a `GUI`.
- To write asynchronous methods with **`async/await`**

Outline

15.1 Introduction

Computer perform operations concurrently

- **In Parallel**

.NET Framework Class Library provides concurrency primitives

- **Namespaces System.Threading, System.Threading.Tasks**

Multithreading

- **“Threads of Execution”**
  - **Each designates a portion of a program executing concurrently**

15.1 Introduction

Performance Tip 15.1

A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can be distributed across multiple processors (if they are available) so that multiple tasks are performed concurrently, allowing the application to operate more efficiently. Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed, another can use the processor.

Good Programming Practice 15.1

Set an object reference to `null` when the program no longer needs that object. This enables the garbage collector to determine at the earliest possible moment that the object can be garbage collected. If such an object has other references to it, that object cannot be collected.

15.2 Thread States: Life Cycle of a Thread

Two critical classes for multithreaded applications: `Thread` **and** `Monitor`

`ThreadStart` **delegate**

- **Specifies the actions the thread will perform during its life cycle**
- **Must be initialized with a method that returns** `void` **and takes no arguments**

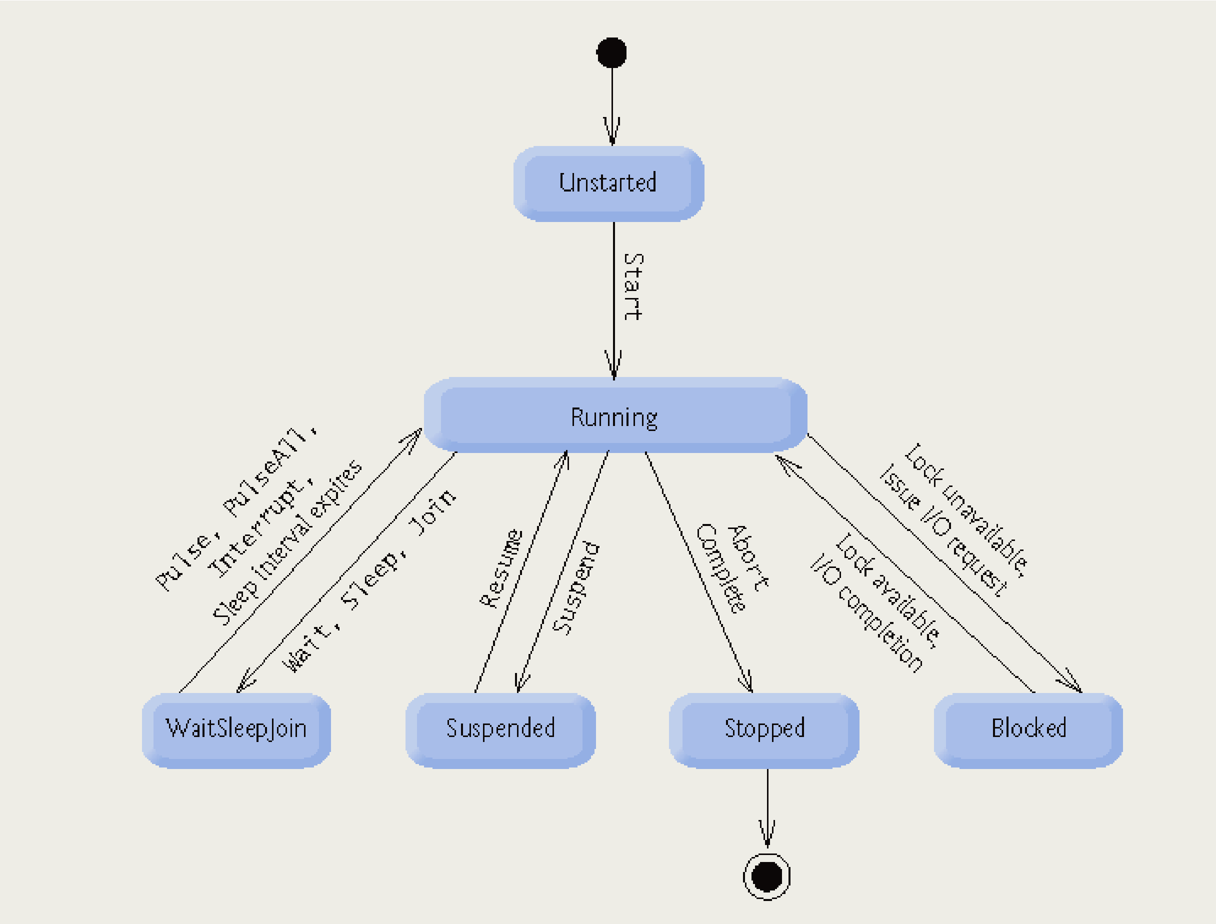


Fig. 15.1 | Thread life cycle.

15.2 Thread States: Life Cycle of a Thread (Cont.)

Thread States

- ***Unstarted* state**
  - **Thread object begins its life cycle in this state**
  - **The thread remains in this state until the program calls the Thread's `Start` method**
    - **This places the thread in the *Running* state**

- ***Running* state**
  - **The thread may not be executing all the time**
    - **Executes only when the operating system assigns a processor to the thread**

- ***Stopped* state**
  - **Enters when `ThreadStart` delegate terminates**
  - **Can force a thread into the *Stopped* state by calling `Thread` method `Abort`**
    - **Abort throws a `ThreadAbortException` in the thread**

15.2 Thread States: Life Cycle of a Thread (Cont.)

- *Blocked* state
  - If a thread is unable to use a processor even if one is available
  - The operating system blocks the thread from executing until all the I/O request for which the thread is waiting is completed
    - At that point, the thread returns to the *Running* state
  - A thread can also becomes blocked because of thread synchronization
    - A thread being synchronized must acquire a lock on an object by calling Monitor method `Enter`
      - If a lock is not available, the thread is blocked until the desired lock becomes available

15.2 Thread States: Life Cycle of a Thread (Cont.)

Three ways in which a *Running* thread enters the *WaitSleepJoin* state

- **A thread encounters code that it cannot execute yet, the thread calls `Monitor` method `Wait`**
  - **Returns to the *Running* state when `Monitor` method `Pulse` or `PulseAll`**
    - **Method `Pulse`**
      - **Moves the next waiting thread back to the *Running* state**
    - **Method `PulseAll`**
      - **Moves all waiting threads back to the *Running* state**
- **A *Running* thread calls Thread method Sleep**
  - **Returns to the *Running* state when its designated sleep time expires**
- **A thread cannot continue executing unless another thread terminates**
  - **That thread calls the other thread's Join method to "join" the two threads**
    - **The dependent thread leaves the *WaitSleepJoin* state and re-enters the *Running* state when the other thread finishes execution**

15.2 Thread States: Life Cycle of a Thread (Cont.)

***Suspended* state**

- **Call thread's `Suspend` method**
  - **Returns to the *Running* state when method `Resume` is called**
- **Methods `Suspend` and `Resume` are now deprecated and should not be used**

***Background* state**

- **`IsBackground` property is set to `True`**
- **A thread can reside in the *Background* state and any other state simultaneously**
- **A process must wait for all foreground threads to finish executing before the process can terminate**
- **The CLR terminates each thread by invoking its `Abort` method if only *Background threads* remains. After all the foreground threads have been stopped, or after the application exits, the system stops all background threads.**

15.3 Thread Priorities and Thread Scheduling

Thread Priorities

- Every thread has a priority in the range between `Lowest` to `Highest`
  - Values are from `ThreadPriority` enumeration
    - `Lowest`, `BelowNormal`, `Normal`, `AboveNormal` and `Highest`
- By default, each new thread has priority `Normal`
- Thread's priority can be adjusted with the `Priority` property
  - An `ArgumentException` occurs, if the value specified is not one of the valid thread-priority constants

15.3 Thread Priorities and Thread Scheduling (Cont.)

Thread scheduler

- **Determine which thread runs next**
- **Simple implementation**
  - **Runs equal-priority threads in a round-robin fashion**
- **Timeslicing**
  - **Each thread receives a brief bust of processor time during which the thread can execute**
    - **Quantum**
- **Higher-priority threads can preempt the currently *running* thread**
  - **Starvation**
    - **Indefinitely postpone lower-priority threads**

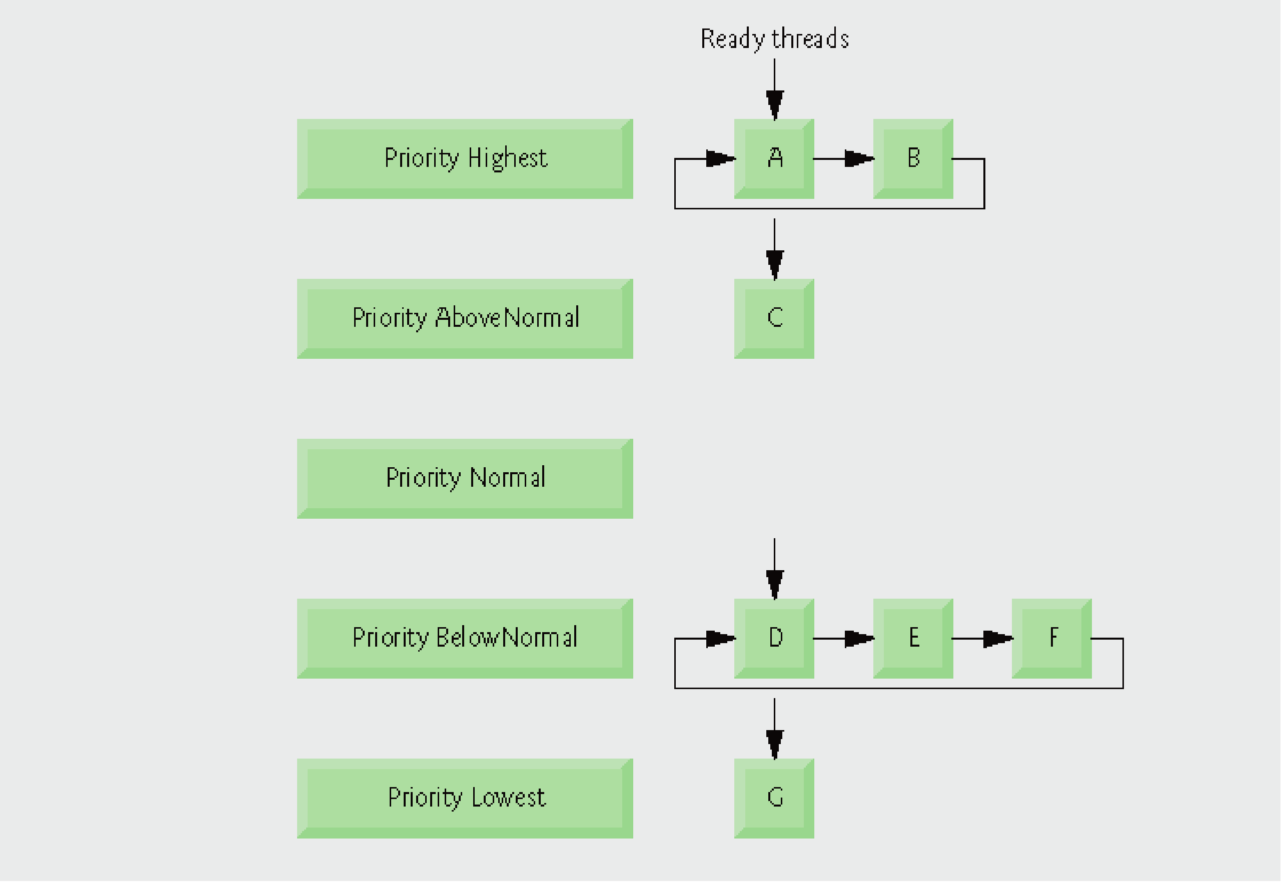


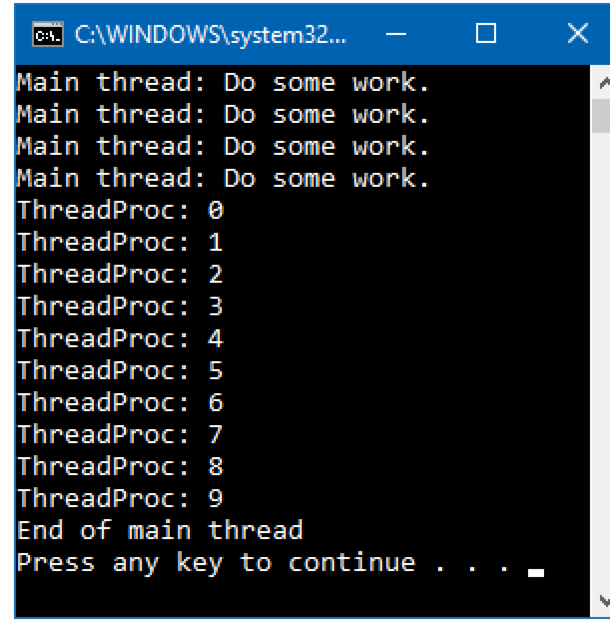
Fig. 15.2 | Thread-priority scheduling.

15.4.1 Creating and Executing Threads

Creating and Executing Threads

- Class `Thread`’s static property `CurrentThread`
  - Returns the current executing thread
- `Thread` property `Name`
  - Specifies the name of the thread
- A `Thread` constructor receives a `ThreadStart` delegate argument
  - The delegate specifies the actions the thread will perform
- During the first time that the system assigns a processor to a thread, the thread calls the method specified by the thread’s `ThreadStart` delegate

```
using System.Threading;

namespace MultiThreading
{
    public static class CreateThread
    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(0);
            }
        }
        public static void Main()
        {
            Thread t = new Thread(new ThreadStart(ThreadMethod));
            t.Start();
            for (int i = 0; i < 4; i++)
            {
                Console.WriteLine("Main thread: Do some work.");
                Thread.Sleep(0);
            }
            t.Join(); // wait for ThreadMethod to end
            Console.WriteLine("End of main thread");
        }
    }
}
```

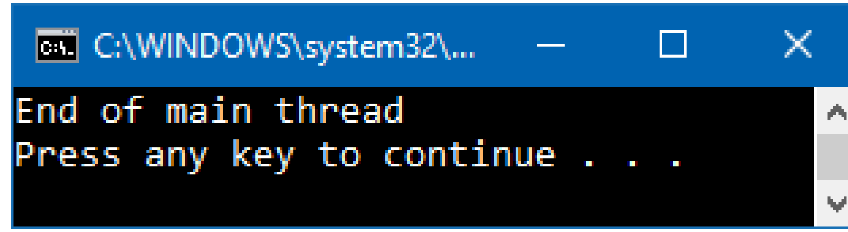


15.4.1 Creating and Executing Threads

This is an example of using the **`Thread`** class to run a method on another thread.

The **`Console`** class synchronizes the use of the output stream for you so you can write to it from multiple threads. **`Synchronization`** is the **mechanism of ensuring that two threads don't execute a specific portion of your program at the same time**. In the case of a console application, this means that no two threads can write data to the screen at the exact same time. If one thread is working with the output stream, other threads will have to wait before it's finished.

As you can see, both threads run and print their message to the console. The **`Thread.Join`** method is called on the main thread to let it wait until **wait until the other thread finishes**.




```
using System;
using System.Threading;

namespace MultiThreading
{
    public static class CreateBackground
    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("ThreadProc: { 0}", i);
                Thread.Sleep(1000);
            }
        }
        public static void Main()
        {
            Thread t = new Thread(ThreadMethod);
            t.IsBackground = true;
            t.Start();
            Console.WriteLine("End of main thread");
        }
    }
}
```

15.4.1 Creating and Executing Threads

It is important to know the difference between *foreground* and *background* threads.

Foreground threads can be used to keep an application alive. Only when all foreground threads end does the common language runtime (CLR) shut down your application. Background threads are then terminated.

If you run the application with the **`IsBackground`** property set to **`true`**, the application exits immediately.

If you set it to **`false`** (creating a foreground thread), the application prints the **`ThreadProc`** message ten times.

Error-Prevention Tip 15.1

Use **`Thread.Sleep(0)`** to signal to Windows that the current thread is finished. Instead of waiting for the whole time-slice of the thread to finish**, it will immediately switch to another thread**. If there's **any other thread** from any process ready to run and **has an equal or higher priority** then **Sleep(0) will yield the processor** and let it run.

Outline

ThreadTester.cs

(1 of 3)

```
// Fig. 15.3: ThreadTester.cs
// Multiple threads printing at different intervals.
using System;
using System.Threading;

// class ThreadTester demonstrates basic threading concepts
class ThreadTester
{
   static void Main( string[] args )
   {
      // Create and name each thread. Use MessagePrinter's
      // Print method as argument to ThreadStart delegate.
      MessagePrinter printer1 = new MessagePrinter();
      Thread thread1 = new Thread ( new ThreadStart( printer1.Print ) );
      thread1.Name = "thread1";

      MessagePrinter printer2 = new MessagePrinter();
      Thread thread2 = new Thread ( new ThreadStart( printer2.Print ) );
      thread2.Name = "thread2";

      MessagePrintersss printer3 = new MessagePrinter();
      Thread thread3 = new Thread ( new ThreadStart( printer3.Print ) );
      thread3.Name = "thread3";

      Console.WriteLine( "Starting threads" );
```

Using the Threading namespace for multithreading

Create a new thread

The task that the thread is to perform

```
      // call each thread's Start method to place each
      // thread in Running state
      thread1.Start();
      thread2.Start();
      thread3.Start();

      Console.WriteLine( "Threads started\n" );
   } // end method Main
} // end class ThreadTester

// Print method of this class used to control threads
class MessagePrinter
{
   private int sleepTime; // passing a value to the thread
   private static Random random = new Random();

   // constructor to initialize a MessagePrinter object
   public MessagePrinter()
   {
      // pick random sleep time between 0 and 5 seconds
      sleepTime = random.Next( 5001 ); // 5001 milliseconds
   } // end constructor

   // method Print controls thread that prints messages
   public void Print()
   {
      // obtain reference to currently executing thread
      Thread current = Thread.CurrentThread;
```

Start each thread

Create `Random` object

Assign a random time for the thread to sleep

`static` method that returns the current running thread

ThreadTester.cs

Outline

```
        // put thread to sleep for sleepTime amount of time
        Console.WriteLine( "{0} going to sleep for {1} milliseconds",
           current.Name, sleepTime );
        Thread.Sleep( sleepTime ); // sleep for sleepTime milliseconds

        // print thread name
        Console.WriteLine( "{0} done sleeping", current.Name );
     } // end method Print
  } // end class MessagePrinter
```

Output how long the thread is sleeping for

ThreadTester.cs

Put the thread to sleep

Output when thread awakens

```
Starting threads
thread1 going to sleep for 1603 milliseconds
thread2 going to sleep for 2355 milliseconds
thread3 going to sleep for 285 milliseconds
Threads started

thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

```
Starting threads
thread1 going to sleep for 4245 milliseconds
thread2 going to sleep for 1466 milliseconds
Threads started

thread3 going to sleep for 1929 milliseconds
thread2 done sleeping
thread3 done sleeping
thread1 done sleeping
```

Error-Prevention Tip 15.1

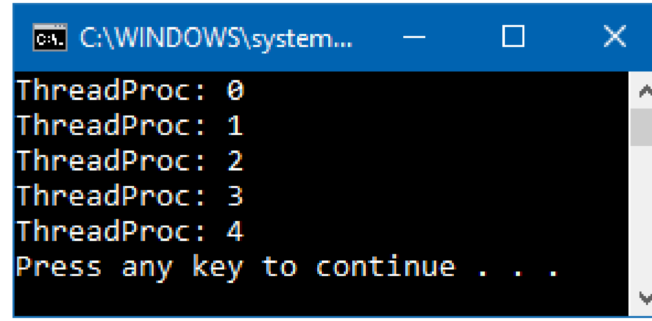
Naming threads helps in the debugging of a multithreaded program. Visual Studio .NET's debugger provides a Threads window that displays the name of each thread and enables you to view the execution of any thread in the program.

15.4.1 Creating and Executing Threads

The **`Thread`** constructor has another overload that takes an instance of a **`ParameterizedThreadStart`** delegate. This overload can be used if you want to pass some data through the start method of your thread to your worker method, as the following slide shows.

In this case, the value **`5`** is passed to the **`ThreadMethod`** as an object. You can **`cast`** it to the expected type to use it in your method.

15.4.1 Creating and Executing Threads




```
class Program
    {
      public static void ThreadMethod(object o)
      {
            for (int i = 0; i < (int)o; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(0);
            }
      }
      public static void Main()
      {
        Thread t =new Thread(new ParameterizedThreadStart(ThreadMethod));
        t.Start(5);
        t.Join();
        }
    }
```

15.4.1 Creating and Executing Threads

A **thread has its own call stack** that stores all the methods that are executed. **Local variables** are stored on the call stack and are **`private`** to the thread.

If you want to **use local data in a thread and initialize it for each thread**, you can use the **`ThreadLocal<T>`** class. This class takes a delegate to a method that initializes the value in each thread.

A **thread can also have its own data** that's **not a local variable**. By marking a field with the **`ThreadStatic`** attribute, **each thread gets its own copy** of a field (see next slide).

```
class Program
    {
        [ThreadStatic]
        public static int  field;
        public static void Main()
        {
            new Thread(() =>
            {
                for (int x = 0; x < 10; x++)
                {
                    field++;
                    Console.WriteLine("Thread A: {0}",
                                                  field);
                }
            }).Start();
            new Thread(() =>
            {
                for (int x = 0; x < 10; x++)
                {
                    field++;
                    Console.WriteLine("Thread B:{0}",
                                                  field);
                }
            }).Start();
            Console.ReadKey();
        }
    }
```

```
C:\WINDOWS\system3...
Thread A: 1
Thread B:1
Thread B:2
Thread B:3
Thread B:4
Thread B:5
Thread B:6
Thread B:7
Thread B:8
Thread B:9
Thread B:10
Thread A: 2
Thread A: 3
Thread A: 4
Thread A: 5
Thread A: 6
Thread A: 7
Thread A: 8
Thread A: 9
Thread A: 10
Press any key to continue . . .
```

```csharp
public static class Program
    {
        public static ThreadLocal<int> _field =
             new ThreadLocal<int>(() =>
             {
                 return Thread.CurrentThread.ManagedThreadId;
             });
        public static void Main()
        {
            new Thread(() =>
            {
                for (int x = 0; x < _field.Value; x++)
                {
                    Console.WriteLine("Thread A: {0}", x);
                }
            }).Start();
            new Thread(() =>
            {
                for (int x = 0; x < _field.Value; x++)
                {
                    Console.WriteLine("Thread B: {0}", x);

                }
            }).Start();
            Console.ReadKey();
        }
    }
```

C:\WINDOWS\syste...

```
Thread A: 0
Thread A: 1
Thread A: 2
Thread B: 0
Thread B: 1
Thread B: 2
Thread B: 3
Press any key to continue . . .
```

Tip

You can use the **`Thread.CurrentThread`** class to ask for information about the thread that's executing. This is called the thread's *execution context*. This property gives you access to properties of the currently executing thread (**`ManagedThreadId`**, thread name, format dates, times, numbers, currency values, the sorting order of text, string comparisons, current security context) and so on.

Tip

The **`Thread`** class isn't something that you should use in your applications, except when you have special needs. However, when using the **`Thread`** class you **have control over all configuration options.** You can, for example, specify the **priority** of your thread, tell Windows that your thread is long running, or configure other advanced options.

15.4.2 Executing a Threadpool

MSDN: Thread pools are often employed in server applications. Each incoming request is assigned to a thread from the thread pool, so the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.

`System.Threading` namespace includes the `ThreadPool` class. This is a `static` class that provides the essential functionality of thread pools. It is useful for running many separate tasks in the background.

15.4.2 Executing a Threadpool

The flow of using the `ThreadPool` looks something like this:

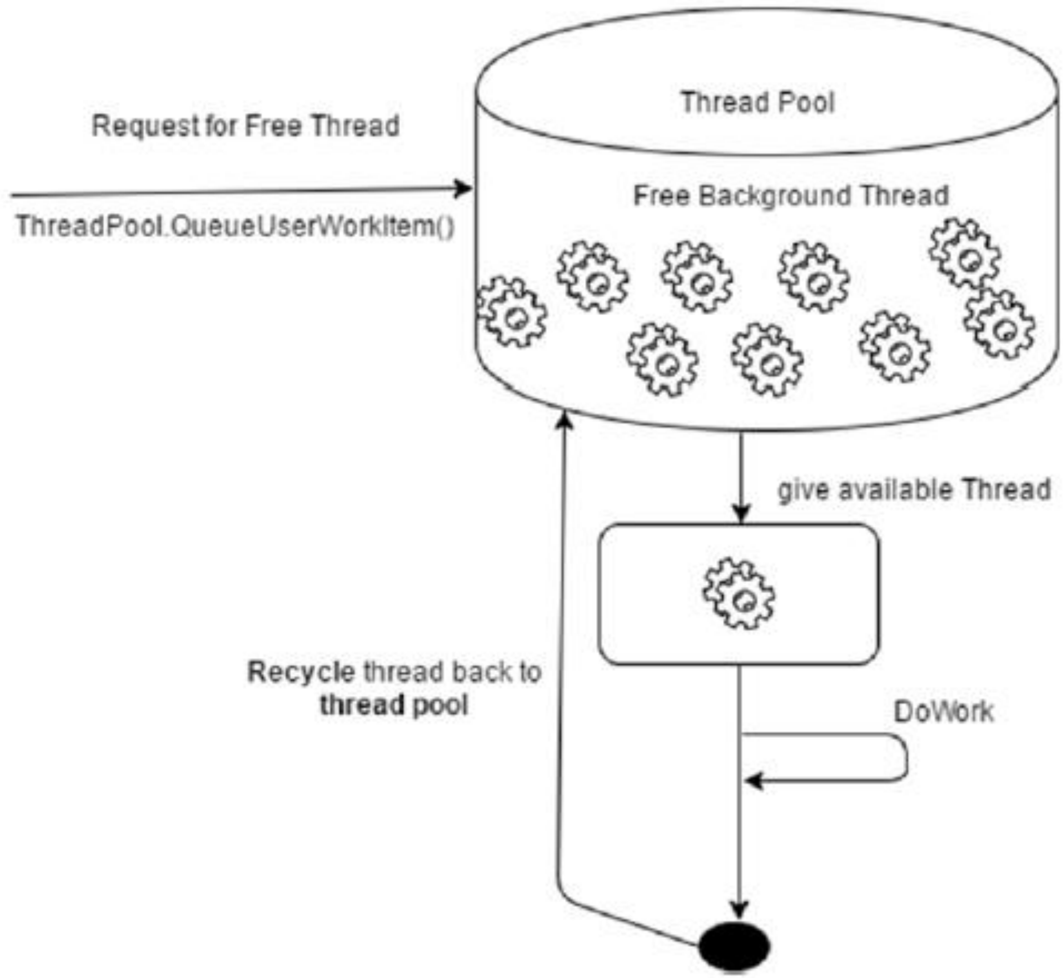
- **You post a method to be executed by the `ThreadPool`**
- **The `ThreadPool` waits for an idle thread**
- **When an idle thread is found the `ThreadPool` uses it to execute your method**
- **Your method executes and terminates**
- **The `ThreadPool` returns the thread and makes it available for other processing**

15.4.2 Executing a Threadpool

A `ThreadPool`, will recycle threads for you, and queue items, if the pool is full. The `ThreadPool` is a *good choice for jobs which are fairly short and frequent*, as it avoids wasting time creating and destroying threads.

15.4.2 Executing a Threadpool

ThreadPool



15.4.2 Executing a Threadpool

The downfall of the `ThreadPool` is that the number of threads is finite and `defaults` to `25` threads `per available processor`. This means that if you queue up `30` tasks, the last five will have to wait for threads to become available from the pool before being executed. To get past the thread count restriction call the `SetMaxThreads` method and pass the number of threads you would like to have available.

15.4.2 Executing a Threadpool

A similar method is `SetMinThreads`.

The `ThreadPool` doesn't really have all of the threads sitting idle and waiting for tasks- it only creates the number of threads that you request `up to` the value given to `SetMaxThreads`. So, if you expect to have the need for, say, 30 threads then you'll want to use `SetMinThreads` to make the minimum number of threads `30`.

15.4.2 Executing a Threadpool

This will increase performance since the `ThreadPool` won't immediately create new threads when needed; it only does this on certain intervals. By `default` this interval is `half of a second`, so the creation of a thread (and delay in your application processing) can be up to half a second `even if you haven't reached the maximum thread threshold`

15.4.2 Executing a Threadpool

Note, also the `ThreadStart` delegate doesn't take any parameters, so in case you have to pass data to the thread execution that information has to be stored somewhere else. One possible way to resolve the problem is creating a new instance of a class, and using that instance to store the information.

15.4.2 Executing a Threadpool

```
public class StringFetcher
{
    private string theString;

    public UrlFetcher (string theString)
    {
        this. theString = theString;
    }

    public void Fetch()
    {
        // use theString here
    }
}

[... in a different class ...]

StringFetcher fetcher = new StringFetcher (myString);
new Thread (new ThreadStart (fetcher.Fetch)).Start();
```

15.4.2 Executing a Threadpool

In some cases, you actually just wish to call a method in some class (possibly the currently executing class) with a specific parameter. In that case, you may wish to use a nested class whose purpose is just to make this call - the state is stored in the class, and the delegate used to start the thread just calls the "real" method with the appropriate parameter.

15.4.2 Executing a Threadpool

A better solution is to call `ThreadPool.QueueUserWorkItem` with a `WaitCallback` `delegate`. One incidental advantage of using a `WaitCallback` is that you can also specify the "`state`" to use when you queue the work item, and this is presented to the `delegate` as a method parameter. It's not strongly typed however (the passed type is just `object`) and it is suggested having a more strongly typed method which is called by the `WaitCallback` delegate.

15.4.2 Executing a Threadpool

```
// You'd use this method elsewhere in code, if you needed it
public void StringFetcher (string theString)
{//the strongly typed method
   this. theString = theString;
}

// You'd use this as the WaitCallback delegate to queue in the thread pool
private void StringFetcher (object theString)
{  // executes in the thread
   StringFetcher ((string) theString);
}

// And run the thread like this, where myString is passed to StringFetcher
ThreadPool.QueueUserWorkItem (new WaitCallback(StringFetcher), myString);
```

15.4.2 Executing a Threadpool

There are many times when you'll need to know when a thread has completed, or get some other information about the status of a thread.

There are several ways to do this, but one of the easiest and most direct is to use `ManualResetEvent` instance within the threads and track the state of this instance from your application.

15.4.2 Executing a Threadpool

In order to find when a thread of a thread pool has completed use a `ManualResetEvent` instance.

1. **Create the `ManualResetEvent` in the main thread as**

```
ManualResetEvent ev =
                    new ManualResetEvent (false);
```

2. **Encapsulate that instance in some `WorkerThread` class that defines the thread method and execute `ManualResetEvent` method `Set()` when the method completes work.**

15.4.2 Executing a Threadpool

```
public class WorkerThread{
   private ManualResetEvent ev;
   // any other data
   public WorkerThread(ManualResetEvent e){
    this.ev= e;
    // any other code here
   }
   public void WaitCallbackMethod(object o){
     // any other code here
      ev.Set(); // the end of the work
   }
}
```

15.4.2 Executing a Threadpool

3. Execute the `WaitCallback` instance in a `ThreadPool`

```
WorkerThread work =
                  new WorkerThread(ev);
ThreadPool.QueueUserWorkItem(
                      work.WaitCallbackMethod,
                  waitCallbackMethodArgumnent);
```

4. Run some more threads in the ThreadPool and store their `ManualResetEvent` in an array `ManualResetEvent[] _events`.

15.4.2 Executing a Threadpool

5. To find when all the threads have completed run

```
WaitHandle.WaitAll( _events);
```

15.4.2 Executing a Threadpool

```
// Fig. 15.3b: ThreadTester.cs
// Multiple threads running in a Thread pool
using System;
using System.Threading;

// class ThreadTester demonstrates basic threading concepts
class ThreadTester
{
    static void Main(string[] args)
    {
        // Create an array of ManualResetEvents for asynch thread pool execution.
        ManualResetEvent[] _events;
        // pick random sleep time between 0 and 5 seconds
        Random random = new Random();

        // Create and name each thread. Use MessagePrinter's
        // Print method as argument to ThreadStart delegate.
        ManualResetEvent e1 = new ManualResetEvent(false);
        MessagePrinter printer1 = new MessagePrinter(e1);
        ThreadPool.QueueUserWorkItem(printer1.Print, random.Next(5001));

        ManualResetEvent e2 = new ManualResetEvent(false);
        MessagePrinter printer2 = new MessagePrinter(e2);
        ThreadPool.QueueUserWorkItem(printer2.Print, random.Next(5001));
```

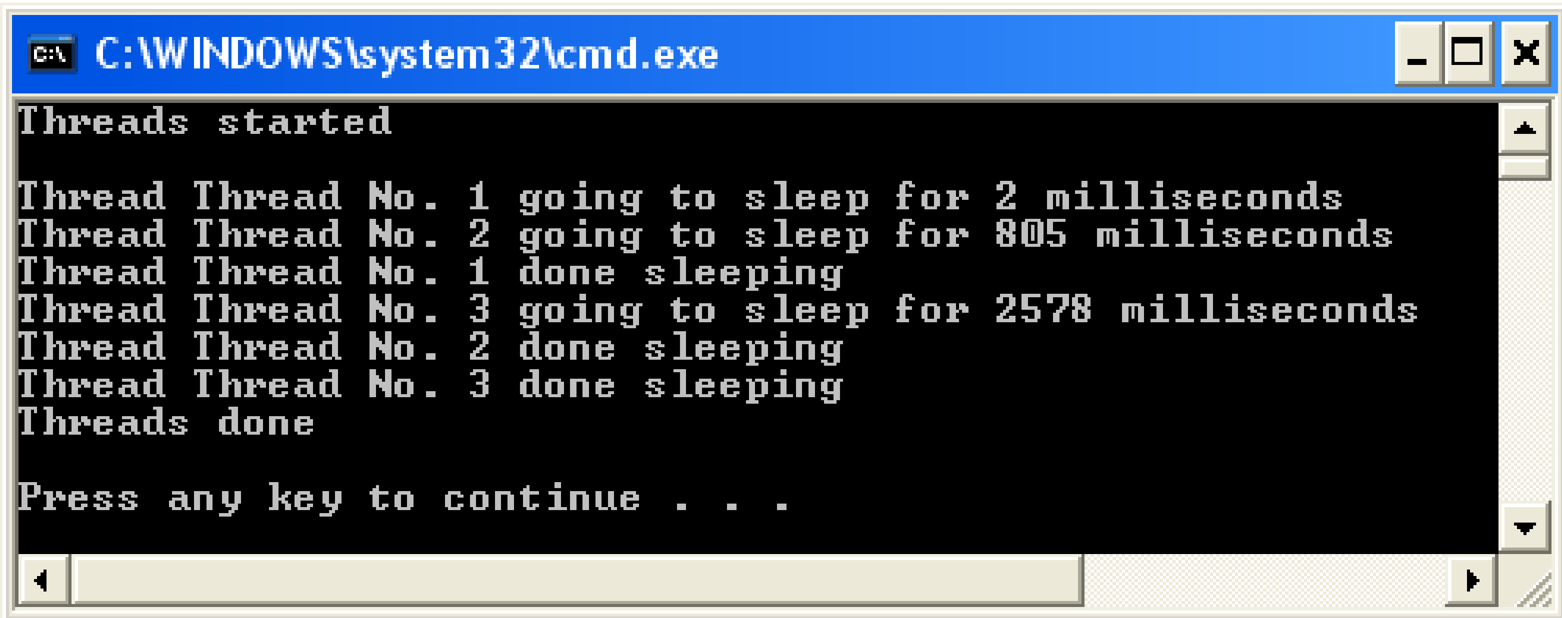
15.4.2 Executing a Threadpool

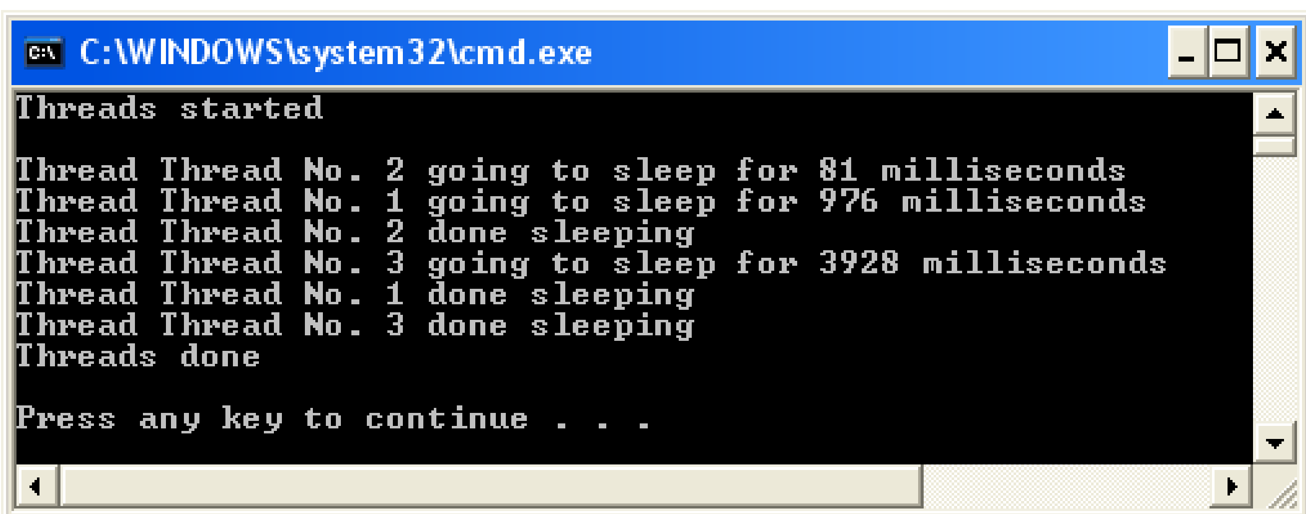
```
        ManualResetEvent e3 = new ManualResetEvent(false);
        MessagePrinter printer3 = new MessagePrinter(e3);
        ThreadPool.QueueUserWorkItem(printer3.Print, random.Next(5001));

        _events = new ManualResetEvent[] { e1, e2, e3 };
        Console.WriteLine("Threads started\n");
        WaitHandle.WaitAll(_events);// wait all threads to complete
        Console.WriteLine("Threads done\n");
    } // end method Main

} // end class ThreadTester

// Print method of this class used to control threads
class MessagePrinter
{

    private static int count = 1;
    private readonly int id;// unique id fo reach thread
    private ManualResetEvent _event;
    // constructor to initialize a MessagePrinter object
    public MessagePrinter(ManualResetEvent _event)
    {
        id = count++;
        this._event = _event;
    } // end constructor
```

15.4.2 Executing a Threadpool

```
    // method Print controls thread that prints messages
    public void Print(object sleepTime)
    {
        // obtain reference to currently executing thread
        Thread current = Thread.CurrentThread;
        current.Name = "Thread No. " + id;
        // put thread to sleep for sleepTime amount of time
        Console.WriteLine("Thread {0} going to sleep for {1} milliseconds",
                                            current.Name, sleepTime);
        Thread.Sleep((int)sleepTime); // sleep for sleepTime milliseconds

        // print thread name
        Console.WriteLine("Thread {0} done sleeping", current.Name);
        _event.Set(); // signal the end of the thread

    } // end method Print
} // end class MessagePrinter
```

15.4.2 Executing a Threadpool





15.5 Thread Synchronization and Class `Monitor`

Problem: (Race conditions determine the output)

- **When multiple threads share data and that data is modified by one or more of those threads, then indeterminate results may occur**

Solution:

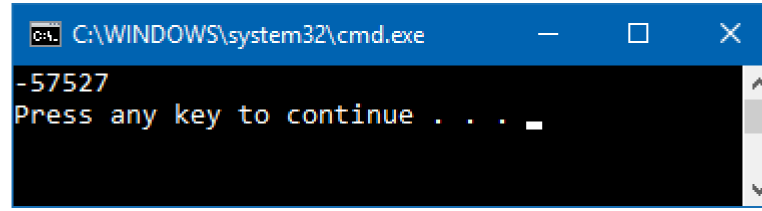
- **Give one thread exclusive access to manipulate the shared data**
- **During that time, other threads wishing to manipulate the data should be kept waiting**
- **When the thread with exclusive access to the data completes its data manipulations, one of the waiting threads should be allowed to proceed**

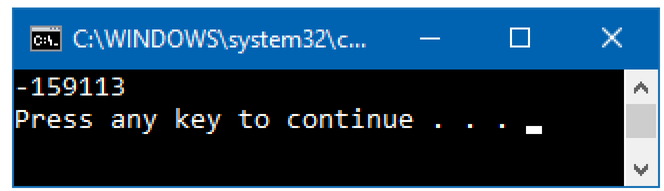
Mutual exclusion or thread synchronization

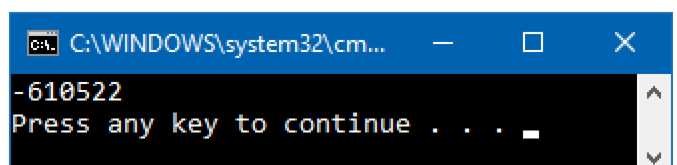
- **Each thread accessing the shared data excludes all other threads from doing so simultaneously**

15.5 Thread Synchronization

```
class Program
    {
        static void Main(string[] args)
        {   // Result depends on the race conditions
            int n = 0;
            var up = new Thread(() =>
                    {

                        for (int i = 0; i < 1000000; i++)
                            n++;
                    });
            up.Start();
            for (int i = 0; i < 1000000; i++)
                n--;
            up.Join();
            Console.WriteLine(n);
        }
    }
```







15.5 Thread Synchronization and Class `Monitor` (Cont.)

This is because **the operation is not *atomic***. It consists of both a read and a write that happen at different moments. This is why access to the **data you're working needs to be *synchronized*, so you can reliably predict how your data is affected**.

It's important to synchronize access to shared data. One feature the C# language offers is the **`lock`** operator, which is some syntactic sugar that the compiler translates in a call to **`System.Thread.Monitor`**.

There are three common ways **to handle synchronization variables** in a **.NET multithreaded environment**.

1. `Lock`
2. `Monitor`
3. `Interlocked`

15.5 Thread Synchronization and Class `Monitor` (Cont.)

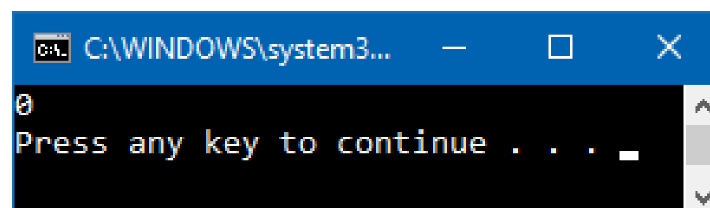
`lock` is a C# operator that **prevents a thread from executing the same block of code that another thread is executing**. **Such a block of code is called a locked code**. We pass the **`lock`** operator **a reference to an object**, which will then be **used to determine** if the **`lock`** applies for the given **block of statements** and to **wait if it is currently in use**. Therefore, if a thread tries to enter a locked code, it will wait until the object is released. When control goes out of the block, the shared memory becomes useable for any thread.

The **`lock`** operator calls **`Monitor.Enter()`** at the start of the block and **`Monitor.Exit()`** at the end of the block.

The best practice is to **use `lock` operator with a private object**, or with a **private static object variable** to protect data common to all instances. This object reference is used in multiple threads, **notifying other threads if someone already used it to lock a block of code**. Hence shared memory becomes thread-safe and the program gives an accurate result.

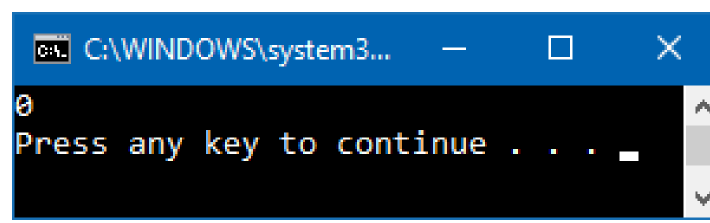
15.5 Thread Synchronization and Class Monitor (Cont.)

```
public static void Main(string[] args)
        { // Result does not depend on the race conditions- Using operator lock
            int n = 0;// shared data between main and worker threads
            object _sharedDataLock = new object();
            var up = new Thread(() =>
            {

                for (int i = 0; i < 1000000; i++)
                    lock (_sharedDataLock) { n++;  // access to shared data is locked
                                             }
                });
            up.Start();
            for (int i = 0; i < 1000000; i++)
                lock (_sharedDataLock) { n--; // access to shared data is locked
                                         }
            up.Join();
            Console.WriteLine(n);
        }
```



15.5 Thread Synchronization and Class Monitor (Cont.)

```
public static void Main(string[] args)
        {   // Result does not depend on the race conditions- Using class Monitor
            int n = 0;// shared data between main and worker threads
            object _sharedData = new object();
            var up = new Thread(() =>
            {

                for (int i = 0; i < 1000000; i++)
                {
                    Monitor.Enter(_sharedData);
                    n++; // access to shared data is locked
                    Monitor.Exit(_sharedData);
                }
            });
            up.Start();
            for (int i = 0; i < 1000000; i++)
            {
                Monitor.Enter(_sharedData);
                n--; // access to shared data is locked
                Monitor.Exit(_sharedData);
            }
            up.Join();
            Console.WriteLine(n);
        }
```



15.5 Thread Synchronization and Class `Monitor` (Cont.)

Making operations atomic is the job of the *Interlocked* class that can be found in the `System.Threading` namespace. When using the `Interlocked.Increment` and `Interlocked.Decrement`, you **create an atomic operation for handling calculations with integers only!**

```
static void Main(string[] args)
        {
            int n = 0;
            var up = Task.Run(() =>
            {
                for (int i = 0; i < 1000000; i++)
                    Interlocked.Increment(ref n); // note the ref keyword!!
            });
            for (int i = 0; i < 1000000; i++)
                Interlocked.Decrement(ref n); // note the ref keyword!!
            up.Wait();
            Console.WriteLine(n);
        }
```

15.5 Thread Synchronization and Class Monitor (Cont.)

```
static void Main(string[] args)
{
        int n = 0;
        var up = Task.Run(() =>
        {
            for (int i = 0; i < 1000000; i++)
                Interlocked.Increment(ref n);
        });
        for (int i = 0; i < 1000000; i++)
            Interlocked.Decrement(ref n);
        up.Wait();
        Console.WriteLine(n);
}
```

15.5 Thread Synchronization and Class `Monitor` (Cont.)

Locking an object allows only one thread to access that object at a time

- **A thread must invoke `Monitor`'s `Enter` method to acquire lock**
  - **Each object has a `SyncBlock` that maintains the state of that object's lock**
  - **Methods of class `Monitor`**
    - **Use the data in an object's `SyncBlock` to determine the state of the lock for that object**
- **A thread can manipulate object's data after acquiring the lock**
- **All threads attempting to acquire the lock on an object that is already locked are blocked**
- **A thread that no longer requires the lock invokes `Monitor` method `Exit` to release the lock**
  - **The `SyncBlock` indicates that the lock for the object is available again**
- **Any threads that were previously blocked from acquiring the lock can acquire the lock to begin its processing of the object**

Common Programming Error 15.1

Make sure that all code that updates a shared object locks the object before doing so. Otherwise, a thread calling a method that does not lock the object can make the object unstable even when another thread has acquired the lock for the object.

Common Programming Error 15.2

Deadlock occurs when a waiting thread (let us call this thread1) cannot proceed because it is waiting (either directly or indirectly) for another thread (let us call this thread2) to proceed, while simultaneously thread2 cannot proceed because it is waiting (either directly or indirectly) for thread1 to proceed. Two threads are waiting for each other, so the actions that would enable either thread to continue execution never occur.

Common Programming Error 15.3

A thread in the *WaitSleepJoin* state cannot re-enter the *Running* state to continue execution until a separate thread invokes `Monitor` method `Pulse` or `PulseAll` with the appropriate object as an argument. If this does not occur, the waiting thread will wait forever—essentially the equivalent of deadlock.

Error-Prevention Tip 15.2

When multiple threads manipulate a shared object using monitors, ensure that if one thread calls `Monitor` method `Wait` to enter the *WaitSleepJoin* state for the shared object, a separate thread eventually will call `Monitor` method `Pulse` to transition the thread waiting on the shared object back to the *Running* state. If multiple threads may be waiting for the shared object, a separate thread can call `Monitor` method `PulseAll` as a safeguard to ensure that all waiting threads have another opportunity to perform their tasks. If this is not done, indefinite postponement or deadlock could occur.

Performance Tip 15.2

Synchronization to achieve correctness in multithreaded programs can make programs run more slowly, as a result of monitor overhead and the frequent transitioning of threads between the *WaitSleepJoin* and *Running* states. There is not much to say, however, for highly efficient, yet incorrect multithreaded programs!

15.6 Producer/Consumer Relationship without Thread Synchronization

Producer/consumer relationship

- Producer generates data and stores it in shared memory
- Consumer reads data from shared memory
- Shared memory is called the buffer

```
// Fig. 15.4: Buffer.cs
// Interface for a shared buffer of int.
using System;

// this interface represents a shared buffer
public interface Buffer
{
   // property Buffer
   int Buffer
   {
      get;
      set;
   } // end property Buffer
 } // end interface Buffer
```

Interface for a buffer

Requires an `int Buffer` property with a `get` and `set` accessor

Outline

Buffer.cs

Outline

Producer.cs

```
// Fig. 15.5: Producer.cs
// Producer produces 10 integer values in the shared buffer.
using System;
using System.Threading;

// class Producer's Produce method controls a thread that
// stores values from 1 to 10 in sharedLocation
public class Producer
{
   private Buffer sharedLocation;
   private Random randomSleepTime;

   // constructor
   public Producer( Buffer shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   } // end constructor
```

Reference an object which implements the `Buffer` interface

Assign `sharedLocation` to the constructor's argument `shared`

Outline

For each iteration, make the producer thread sleep for 0-3 seconds

Producer.cs

Assign `count` to `sharedLocation`'s `Buffer` property

```
   // store values 1-10 in object sharedLocation
   public void Produce()
   {
      // sleep for random interval up to 3000 milliseconds
      // then set sharedLocation's Buffer property
      for ( int count = 1; count <= 10; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3001 ) );
         sharedLocation.Buffer = count;
      } // end for

      Console.WriteLine( "{0} done producing.\nTerminating {0}.",
         Thread.CurrentThread.Name );
   } // end method Produce
} // end class Producer
```

Output to notify that the producer has finished

Consumer.cs

```
// Fig. 15.6: Consumer.cs
// Consumer consumes 10 integer values from the shared buffer.
using System;
using System.Threading;

// class Consumer's Consume method controls a thread that
// loops 10 times and reads a value from sharedLocation
public class Consumer
{
   private Buffer sharedLocation;
   private Random randomSleepTime;

   // constructor
   public Consumer( Buffer shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   } // end constructor
```

Reference an object which implements the `Buffer` interface

Assign `sharedLocation` to the constructor's argument `shared`

Consumer.cs

```
    // read sharedLocation's value ten times
    public void Consume()
    {
       int sum = 0;

       // sleep for random interval up to 3000 milliseconds then
       // add sharedLocation's Buffer property value to sum
       for ( int count = 1; count <= 10; count++ )
       {
          Thread.Sleep( randomSleepTime.Next( 1, 3001 ) );
          sum += sharedLocation.Buffer;
       } // end for

       Console.WriteLine(
          "{0} read values totaling: {1}.\nTerminating {0}.",
          Thread.CurrentThread.Name, sum );
    } // end method Consume
} // end class Consumer
```

Local variable to sum numbers from buffer

For each iteration, make the consumer thread sleep for 0-3 seconds

Output results

Outline

UnsynchronizedBuffer.cs

```
// Fig. 15.7: UnsynchronizedBuffer.cs
// An unsynchronized shared buffer implementation.
using System;
using System.Threading;

// this class represents a single shared int
public class UnsynchronizedBuffer : Buffer
{
   // buffer shared by producer and consumer threads
   private int buffer = -1;

   // property Buffer
   public int Buffer
   {
      get
      {
         Console.WriteLine( "{0} reads {1}",
            Thread.CurrentThread.Name, buffer );
         return buffer;
      } // end get
      set
      {
         Console.WriteLine( "{0} writes {1}",
            Thread.CurrentThread.Name, value );
         buffer = value;
      } // end set
   } // end property Buffer
} // end class UnsynchronizedBuffer
```

Implements from interface `Buffer`

`int` instance variable which represents the buffer shared by producers and consumers

Declare `int Buffer` property to satisfy `Buffer` interface's requirements

Output and return buffer value for the consumer

Output and set buffer value for the producer

Outline

UnsynchronizedBufferTest.cs

```
// Fig. 15.8: UnsynchronizedBufferTest.cs
// Showing multiple threads modifying a shared object without
// synchronization.
using System;
using System.Threading;

// this class creates producer and consumer threads
class UnsynchronizedBufferTest
{
   // create producer and consumer threads and start them
   static void Main( string[] args )
   {
      // create shared object used by threads
      UnsynchronizedBuffer shared = new UnsynchronizedBuffer();

      // Random object used by each thread
      Random random = new Random();
```

Create an `UnsynchronizedBuffer` object for the producer and consumer

Outline

UnsynchronizedBufferTest.cs

```

      // create Producer and Consumer objects
      Producer producer = new Producer( shared, random );
      Consumer consumer = new Consumer( shared, random );

      // create threads for producer and consumer and set
      // delegates for each thread
      Thread producerThread =
         new Thread( new ThreadStart( producer.Produce ) );
      producerThread.Name = "Producer";

      Thread consumerThread =
         new Thread( new ThreadStart( consumer.Consume ) );
      consumerThread.Name = "Consumer";

      // start each thread
      producerThread.Start();
      consumerThread.Start();
   } // end Main
} // end class UnsynchronizedBufferTest
```

Create the producer and consumer

Create the threads for the producer and consumer

The action to be performed by the threads

Start the producer and consumer threads

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Consumer reads 3
Producer writes 4
Consumer reads 4
Producer writes 5
Consumer reads 5
Consumer reads 5
Producer writes 6
Consumer reads 6
Consumer read values totaling: 30.
Terminating Consumer.
Producer writes 7
Producer writes 8
Producer writes 9
Producer writes 10
Producer done producing.
Terminating Producer.
```


UnsynchronizedBufferTest.cs

(3 of 4)

Outline

```
Producer writes 1
Producer writes 2
Consumer reads 2
Consumer reads 2
Producer writes 3
Producer writes 4
Consumer reads 4
Consumer reads 4
Producer writes 5
Consumer reads 5
Producer writes 6
Consumer reads 6
Producer writes 7
Consumer reads 7
Producer writes 8
Consumer reads 8
Producer writes 9
Producer writes 10
Producer done producing.
Terminating Producer.
Consumer reads 10
Consumer reads 10
Consumer read values totaling: 58.
Terminating Consumer.
```

UnsynchronizedBufferTest.cs

(4 of 4)

15.7 Producer/Consumer Relationship with Thread Synchronization

Producer/consumer relationship

- This example uses class `Monitor` to implement synchronization

Outline

```
// Fig. 15.9: SynchronizedBuffer.cs
// A synchronized shared buffer implementation.
using System;
using System.Threading;

// this class represents a single shared int
public class SynchronizedBuffer : Buffer
{
   // buffer shared by producer and consumer threads
   private int buffer = -1;

   // occupiedBufferCount maintains count of occupied buffers
   private int occupiedBufferCount = 0;

   // property Buffer
   public int Buffer
   {
      get
      {
         // obtain lock on this object
         Monitor.Enter( this );
```

Implements from interface `Buffer`

SynchronizedBuffer.cs

`int` instance variable which represents the buffer shared by producers and consumers

Counter to control access to the buffer

Declare `int Buffer` property to satisfy `Buffer` interface's requirements

Acquire lock on the `SynchronizedBuffer` object

```
      // if there is no data to read, place invoking
      // thread in WaitSleepJoin state
      while ( occupiedBufferCount == 0 )
      {
         Console.WriteLine(
            Thread.CurrentThread.Name + " tries to read." );
         DisplayState( "Buffer empty. " +
            Thread.CurrentThread.Name + " waits." );
         Monitor.Wait( this ); // enter WaitSleepJoin state
      } // end if

      // indicate that producer can store another value
      // because consumer is about to retrieve a buffer value
      --occupiedBufferCount;

      DisplayState( Thread.CurrentThread.Name + " reads " + buffer );

      // tell waiting thread (if there is one) to
      // become ready to execute (Running state)
      Monitor.Pulse( this );
```

The consumer checks to see if it can "consume"

The consumer waits until the producer "produces"

Adjust counter so producer can continue "producing"

Output results

Allow producer to continue

Outline

SynchronizedBuffer.cs

(3 of 5)

```
         // Get copy of buffer before releasing lock.
         // It is possible that the producer could be
         // assigned the processor immediately after the
         // monitor is released and before the return
         // statement executes. In this case, the producer
         // would assign a new value to buffer before the
         // return statement returns the value to the
         // consumer. Thus, the consumer would receive the
         // new value. Making a copy of buffer and
         // returning the copy ensures that the
         // consumer receives the proper value.
         int bufferCopy = buffer;

         // release lock on this object
         Monitor.Exit( this );

         return bufferCopy;
      } // end get
      set
      {
         // acquire lock for this object
         Monitor.Enter( this );
```

Release lock on the `SynchronizedBuffer` object

Return the buffer's value

Acquire lock on the `SynchronizedBuffer` object

Outline

SynchronizedBuffer.cs

(4 of 5)

```
        // if there are no empty locations, place invoking
        // thread in WaitSleepJoin state
        while ( occupiedBufferCount == 1 )
        {
           Console.WriteLine(
              Thread.CurrentThread.Name + " tries to write." );
           DisplayState( "Buffer full. " +
              Thread.CurrentThread.Name + " waits." );
           Monitor.Wait( this ); // enter WaitSleepJoin state
        } // end if

        // set new buffer value
        buffer = value;

        // indicate consumer can retrieve another value
        // because producer has just stored a buffer value
        ++occupiedBufferCount;

        DisplayState( Thread.CurrentThread.Name + " writes " + buffer );

        // tell waiting thread (if there is one) to
        // become ready to execute (Running state)
        Monitor.Pulse( this );
```

The producer checks to see if it can "produce"

The producer waits until the consumer "consumes"

The producer sets buffer's new value

Adjust counter so consumer can continue "consuming"

Output results

Allow consumer to continue

Outline

```
        // release lock on this object
        Monitor.Exit( this );
     } // end set
  } // end property Buffer

  // display current operation and buffer state
  public void DisplayState( string operation )
  {
     Console.WriteLine( "{0,-35}{1,-9}{2}\n",
        operation, buffer, occupiedBufferCount );
  } // end method DisplayState
} // end class SynchronizedBuffer
```

Release lock on the `SynchronizedBuffer` object

Output results

SynchronizedBuffer.cs

(5 of 5)

Common Programming Error 15.4

Forgetting to release the lock on an object when that lock is no longer needed is a logic error. This will prevent the threads in your program from acquiring the lock to proceed with their tasks. These threads will be forced to wait (unnecessarily, because the lock is no longer needed). Such waiting can lead to deadlock and indefinite postponement.

Outline

SynchronizedBufferTest.cs

```
// Fig. 15.10: SynchronizedBufferTest.cs
// Showing multiple threads modifying a shared object with
// synchronization.
using System;
using System.Threading;

// this class creates producer and consumer threads
class SynchronizedBufferTest
{
   // create producer and consumer threads and start them
   static void Main( string[] args )
   {
      // create shared object used by threads
      SynchronizedBuffer shared = new SynchronizedBuffer();

      // Random object used by each thread
      Random random = new Random();

      // output column heads and initial buffer state
      Console.WriteLine( "{0,-35}{1,-9}{2}\n",
         "Operation", "Buffer", "Occupied Count" );
      shared.DisplayState( "Initial state" );
```

Create a `SynchronizedBuffer` object for the producer and consumer

Output table header and initial information

Outline

SynchronizedBufferTest.cs

```
      // create Producer and Consumer objects
      Producer producer = new Producer( shared, random );
      Consumer consumer = new Consumer( shared, random );

      // create threads for producer and consumer and set
      // delegates for each thread
      Thread producerThread =
         new Thread( new ThreadStart( producer.Produce ) );
      producerThread.Name = "Producer";

      Thread consumerThread =
         new Thread( new ThreadStart( consumer.Consume ) );
      consumerThread.Name = "Consumer";

      // start each thread
      producerThread.Start();
      consumerThread.Start();
   } // end Main
} // end class SynchronizedBufferTest
```

Create the producer and consumer

Create the threads for the producer and consumer

The action to be performed by the threads

Start the producer and consumer threads

Outline

SynchronizedBufferTest.cs

(3 of 6)

```
Operation                           Buffer   Occupied Count
Initial state                       -1       0
Producer writes 1                   1        1
Consumer reads 1                    1        0
Consumer tries to read.
Buffer empty. Consumer waits.       1        0
Producer writes 2                   2        1
Consumer reads 2                    2        0
Producer writes 3                   3        1
Producer tries to write.
Buffer full. Producer waits.        3        1
Consumer reads 3                    3        0
Producer writes 4                   4        1
Consumer reads 4                    4        0
Consumer tries to read.
Buffer empty. Consumer waits.       4        0
Producer writes 5                   5        1
Consumer reads 5                    5        0
Producer writes 6                   6        1
```

(continued)

Outline

SynchronizedBufferTest.cs

(4 of 6)

```
Consumer reads 6                  6       0
Producer writes 7                 7       1
Consumer reads 7                  7       0
Producer writes 8                 8       1
Producer tries to write.
Buffer full. Producer waits.      8       1
Consumer reads 8                  8       0
Producer writes 9                 9       1
Consumer reads 9                  9       0
Consumer tries to read.
Buffer empty. Consumer waits.     9       0
Producer writes 10                10      1
Producer done producing.
Terminating Producer.
Consumer reads 10                 10      0
Consumer read values totaling: 55.
Terminating Consumer.
```

Outline

SynchronizedBufferTest.cs

(5 of 6)

```
Operation                           Buffer   Occupied Count
Initial state                       -1       0

Consumer tries to read.
Buffer empty. Consumer waits.       -1       0

Producer writes 1                   1        1

Consumer reads 1                    1        0

Consumer tries to read.
Buffer empty. Consumer waits.       1        0

Producer writes 2                   2        1

Consumer reads 2                    2        0

Producer writes 3                   3        1

Consumer reads 3                    3        0
```

(continued)

```
Producer writes 4                    4        1
Producer tries to write.
Buffer full. Producer waits.         4        1
Consumer reads 4                     4        0
Producer writes 5                    5        1
Producer tries to write.
Buffer full. Producer waits.         5        1
Consumer reads 5                     5        0
Producer writes 6                    6        1
Consumer reads 6                     6        0
Producer writes 7                    7        1
Consumer reads 7                     7        0
Producer writes 8                    8        1
Consumer reads 8                     8        0
Consumer tries to read.
Buffer empty. Consumer waits.        8        0
Producer writes 9                    9        1
Consumer reads 9                     9        0
Consumer tries to read.
Buffer empty. Consumer waits.        9        0
Producer writes 10                   10       1
Consumer reads 10                    10       0
Producer done producing.
Terminating Producer.
Consumer read values totaling: 55.
Terminating Consumer.
```


SynchronizedBufferTest.cs

(6 of 6)

15.8 Producer/Consumer Relationship Circular Buffer

Circular buffer

- Provides extra buffer space into which producer can place values and consumer can read values
  - If the buffer is too large, it will waste memory
  - Define circular buffer with enough extra cells to handle the anticipate “extra” production
- Minimize the waiting for threads to share resources and operate at relative speeds
  - Inappropriate if the producer and consumer operate at different speeds
- This example uses class `locks` to implement synchronization

Performance Tip 15.3

Even when using a circular buffer, it is possible that a producer thread could fill the buffer, which would force the producer thread to wait until a consumer consumes a value to free an element in the buffer. Similarly, if the buffer is empty, the consumer thread must wait until the producer produces another value. The key to using a circular buffer is optimizing the buffer size to minimize the amount of thread wait time.

Common Programming Error 15.5

When using class `Monitor's` `Enter` and `Exit` methods to manage an object's lock, `Exit` must be called explicitly to release the lock. If an exception occurs in a method before `Exit` can be called and that exception is not caught, the method could terminate without calling `Exit`. If so, the lock is not released. To avoid this error, place code that could throw exceptions in a `try` block, and place the call to `Exit` in the corresponding `finally` block to ensure that the lock is released.

Software Engineering Observation 15.1

Using a `lock` block to manage the lock on a synchronized object eliminates the possibility of forgetting to relinquish the lock with a call to `Monitor` method `Exit`. C# implicitly calls `Monitor` method `Exit` when a `lock` block terminates for any reason. Thus, even if an exception occurs in the block, the lock will be released.

Outline

CircularBuffer.cs

```
// Fig. 15.11: CircularBuffer.cs
// A circular shared buffer for the producer/consumer relationship.
using System;
using System.Threading;

// implement the an array of shared integers with synchronization
public class CircularBuffer : Buffer
{
   // each array element is a buffer
   private int[] buffers = { -1, -1, -1 };

   // occupiedBufferCount maintains count of occupied buffers
   private int occupiedBufferCount = 0;

   private int readLocation = 0; // location of the next read
   private int writeLocation = 0; // location of the next write
```

Implements the `Buffer` interface

Represent a circular buffer

`private` instance variable for reading and writing

```
// property Buffer
public int Buffer
{
   get
   {
      // lock this object while getting value
      // from buffers array
      lock ( this )
      {
         // if there is no data to read, place invoking
         // thread in WaitSleepJoin state
         while ( occupiedBufferCount == 0 )
         {
            Console.Write( "\nAll buffers empty. {0} waits.",
               Thread.CurrentThread.Name );
            Monitor.Wait( this ); // enter the WaitSleepJoin state
         } // end if

         // obtain value at current readLocation
         int readValue = buffers[ readLocation ];

         Console.Write( "\n{0} reads {1} ",
            Thread.CurrentThread.Name, buffers[ readLocation ] );

         // just consumed a value, so decrement number of
         // occupied buffers
         --occupiedBufferCount;
```

Locks this object

Read data from the circular buffer

Output the read value

Adjust number of occupied buffer space

```
            // update readLocation for future read operation,
            // then add current state to output
            readLocation = ( readLocation + 1 ) % buffers.Length;
            Console.Write( CreateStateOutput() );

            // return waiting thread (if there is one)
            // to Running state
            Monitor.Pulse( this );

            return readValue;
         } // end lock
      } // end get
      set
      {
         // lock this object while setting value
         // in buffers array
         lock ( this )
         {
            // if there are no empty locations, place invoking
            // thread in WaitSleepJoin state
            while ( occupiedBufferCount == buffers.Length )
            {
               Console.Write( "\nAll buffers full. {0} waits.",
                  Thread.CurrentThread.Name );
               Monitor.Wait( this ); // enter the WaitSleepJoin state
            } // end if
```

Specify which location to read from buffer for the next consumer

CircularBuffer.cs

Output the locations of the reader and writer

Locks this object

```
            // place value in writeLocation of buffers
            buffers[ writeLocation ] = value;

            Console.Write( "\n{0} writes {1} ",
               Thread.CurrentThread.Name, buffers[ writeLocation ] );

            // just produced a value, so increment number of
            // occupied buffers
            ++occupiedBufferCount;

            // update writeLocation for future write operation,
            // then add current state to output
            writeLocation = ( writeLocation + 1 ) % buffers.Length;
            Console.Write( CreateStateOutput() );

            // return waiting thread (if there is one)
            // to Running state
            Monitor.Pulse( this );
         } // end lock
```

Write new value to the circular buffer

Output the new value

Adjust number of occupied buffer space

Specify which location to write to next

Output the locations of the reader and writer

CircularBuffer.cs

```
      } // end set
   } // end property Buffer

   // create state output
   public string CreateStateOutput()
   {
      // display first line of state information
      string output = "(buffers occupied: " +
         occupiedBufferCount + ")\nbuffers: ";

      for ( int i = 0; i < buffers.Length; i++ )
         output += " " + string.Format( "{0,2}", buffers[ i ] ) + "  ";

      output += "\n";

      // display second line of state information
      output += "         ";

      for ( int i = 0; i < buffers.Length; i++ )
         output += "---- ";

      output += "\n";

```

Add the number of occupied spaces to the `string`

Add the values of the buffer to the `string`

```
      // display third line of state information
      output += "          ";

      // display readLocation (R) and writeLocation (W)
      // indicators below appropriate buffer locations
      for ( int i = 0; i < buffers.Length; i++ )
      {
         if ( i == writeLocation &&
            writeLocation == readLocation )
            output += " WR  ";
         else if ( i == writeLocation )
            output += " W   ";
         else if  ( i == readLocation )
            output += "  R  ";
         else
            output += "     ";
      } // end for

      output += "\n";
      return output;
   } // end method CreateStateOutput
} // end class HoldIntegerSynchronized
```

Add the position of the reader and writer to the `string`

CircularBuffer.cs

(6 of 6)

Outline

CircularBufferTest.cs

```
// Fig. 15.12: CircularBufferTest.cs
// Implementing the producer/consumer relationship with a
// circular buffer.
using System;
using System.Threading;

class CircularBufferTest
{
   // create producer and consumer threads and start them
   static void Main( string[] args )
   {
      // create shared object used by threads
      CircularBuffer shared = new CircularBuffer();

      // Random object used by each thread
      Random random = new Random();

      // display shared state before producer
      // and consumer threads begin execution
      Console.Write( shared.CreateStateOutput() );

      // create Producer and Consumer objects
      Producer producer = new Producer( shared, random );
      Consumer consumer = new Consumer( shared, random );
```

Create a `CircularBuffer` object for the producer and consumer

Output initial information

Create the producer and consumer

```
        // create threads for producer and consumer and set
        // delegates for each thread
        Thread producerThread =
           new Thread( new ThreadStart( producer.Produce ) );
        producerThread.Name = "Producer";

        Thread consumerThread =
           new Thread( new ThreadStart( consumer.Consume ) );
        consumerThread.Name = "Consumer";

        // start each thread
        producerThread.Start();
        consumerThread.Start();
     } // end Main
  } // end class CircularBufferTest
```

Create the threads for the producer and consumer

t

The action to be performed by the threads

Start the producer and consumer threads

Outline

CircularBufferTest.cs

(3 of 6)

```
(buffers occupied: 0)
buffers:  -1   -1   -1
         ---- ---- ----
          WR

All buffers empty. Consumer waits.
Producer writes 1 (buffers occupied: 1)
buffers:   1   -1   -1
         ---- ---- ----
           R    W

Consumer reads 1 (buffers occupied: 0)
buffers:   1   -1   -1
         ---- ---- ----
               WR

Producer writes 2 (buffers occupied: 1)
buffers:   1    2   -1
         ---- ---- ----
                R    W

Consumer reads 2 (buffers occupied: 0)
buffers:   1    2   -1
         ---- ---- ----
                    WR

All buffers empty. Consumer waits.
Producer writes 3 (buffers occupied: 1)
buffers:   1    2    3
         ---- ---- ----
          W          R

Consumer reads 3 (buffers occupied: 0)
buffers:   1    2    3
         ---- ---- ----
          WR
```

```
All buffers empty. Consumer waits.
Producer writes 4 (buffers occupied: 1)
buffers:   4    2    3
         ---- ---- ----
           R   W

Producer writes 5 (buffers occupied: 2)
buffers:   4    5    3
         ---- ---- ----
           R        W

Consumer reads 4 (buffers occupied: 1)
buffers:   4    5    3
         ---- ---- ----
                R   W

Producer writes 6 (buffers occupied: 2)
buffers:   4    5    6
         ---- ---- ----
          W     R

Producer writes 7 (buffers occupied: 3)
buffers:   7    5    6
         ---- ---- ----
               WR
```

Outline

CircularBufferTest.cs

(4 of 6)

Outline

```
All buffers full. Producer waits.
Consumer reads 5 (buffers occupied: 2)
buffers:   7    5    6
         ---- ---- ----
               W     R

Consumer reads 6 (buffers occupied: 1)
buffers:   7    5    6
         ---- ---- ----
           R   W

Producer writes 8 (buffers occupied: 2)
buffers:   7    8    6
         ---- ---- ----
           R        W

Consumer reads 7 (buffers occupied: 1)
buffers:   7    8    6
         ---- ---- ----
                R   W

Consumer reads 8 (buffers occupied: 0)
buffers:   7    8    6
         ---- ---- ----
                    WR

Producer writes 9 (buffers occupied: 1)
buffers:   7    8    9
         ---- ---- ----
          W          R
```

CircularBufferTest.cs

(5 of 6)

```
Producer writes 10 (buffers occupied: 2)
buffers:  10    8    9
         ---- ---- ----
               W     R
Producer done producing.
Terminating Producer.

Consumer reads 9 (buffers occupied: 1)
buffers:  10    8    9
         ---- ---- ----
           R   W

Consumer reads 10 (buffers occupied: 0)
buffers:  10    8    9
         ---- ---- ----
               WR
Consumer read values totaling: 55.
Terminating Consumer.
```

Outline

CircularBufferTest.cs

(6 of 6)

15.9 Multithreading with GUIs

WPF and Windows form components

- **Not thread safe**
- **All interactions with GUI should be performed by the User Interface thread**

GUI objects have *thread affinity* , meaning that they belong to a particular thread. Your code must always use a user interface element on the same thread that created it. It is illegal to attempt to use any GUI element from any other thread.

Windows Forms use class `Control`'s method `Invoke` to specify GUI processing statements that UI thread should execute.

15.9 Multithreading with GUIs

WPF uses this model for various reasons.

One is simplicity- **the model is straightforward and does not introduce any complications** for applications that have no need for multiple threads. This **simplicity** also makes it fairly straightforward for WPF to detect when you have broken the rules. Another important reason for using a single-threaded model is to support interop with Win32, which also has thread affinity requirements. By adopting a **strict thread affinity model**, you can mix WPF, Windows Forms, and Win32 user interface elements freely within a single application.

15.9 Multithreading with GUIs

WPF types with **thread affinity** derive from the **`DispatcherObject`** base class. **`Brush`** and **`Geometry`** both derive from **`DispatcherObject`**, so usually you can use them only on the thread that created them. **`Color`** does not, and therefore you can use it on a different thread from the one on which it was created. **`DispatcherObject`** provides a couple of methods that let you check whether you are on the right thread for the object: **`CheckAccess()`** and **`VerifyAccess()`**.

`CheckAccess` returns **`true`** if you are on the correct thread, **`false`** otherwise.

`VerifyAccess` is intended for when you think you are already on the **right thread**, and it would be indicative of a problem in the program if you were not. It **`throws`** an **`exception,`** if you are on the **wrong thread**.

15.9 Multithreading with GUIs

Obtaining a Dispatcher in WPF

All WPF objects with thread affinity derive from the **`DispatcherObject`** base class. This class defines a **`Dispatcher`** property, which returns the **`Dispatcher`** object for the thread to which the object belongs.

You can also retrieve the **`Dispatcher`** for the current thread by using the **`Dispatcher.CurrentDispatcher`** static property.

The **`Dispatcher`** provides methods that let you invoke the code of your choice on the dispatcher's thread. You can use either **`Invoke`** or **`BeginInvoke`**. Both of these **accept any delegate and an optional list of parameters**. They both invoke the delegate's target method on the dispatcher's thread, regardless of which thread you call them from.

`Invoke` does not return until the method has been executed, whereas **`BeginInvoke`** queues the request to invoke the method, but **returns straight away without waiting** for the method to run. It is an **asynchronous version** of Invoke.

When **`wpfControlRef.Dispatcher.CheckAccess()`** returns **`true`** then **you can directly access** **`wpfControlRef`** **and its properties** because the dispatcher's thread (the UI thread) is the currently running thread.

15.9 Multithreading with GUIs

```
partial class MyWindow : Window
    {
        //..A typical example in WPF
        // how to marshal a request back to the UI thread.
        public void SomeMethod(Color bgColor) {
            this.Background = new SolidColorBrush(bgColor);
        }
        void RunsOnWorkerThread() // runs on a worker thread
        {
            Color bgColor = Color.FromRgb(255, 0, 0);
            //call Dispatcher to change a property of a WPF
            // component
            this.Dispatcher.BeginInvoke(
                                DispatcherPriority.Normal,
                                new Action<Color>(SomeMethod),
                                bgColor);
        }
        //...
    }
```

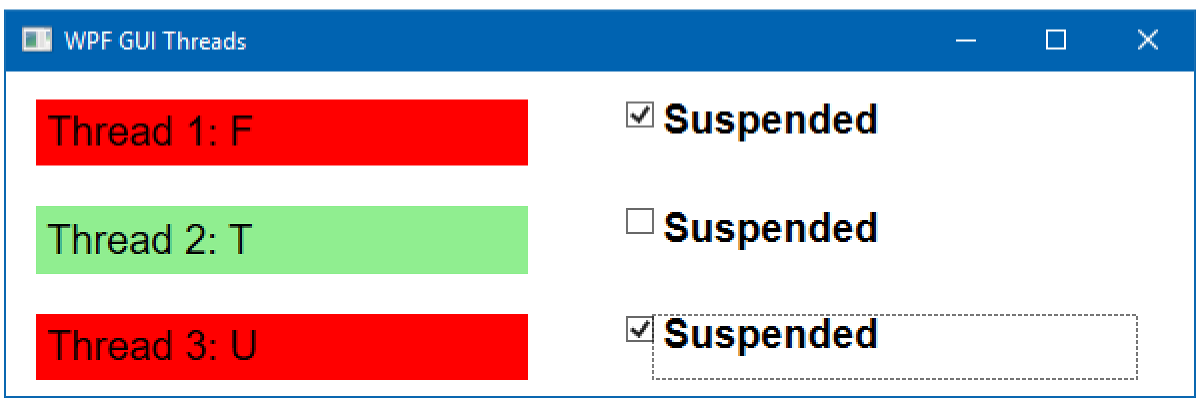
15.9 Multithreading with GUIs

Synchronizing the UI thread with worker threads in WPF.

Obtaining a Dispatcher in WPF

The following WPF application shows how to:

- synchronize a worker thread with the UI thread
- start and stop a worker thread
- distinguish code running in the UI thread and the worker thread



```
// Writes a random letter to a WPF label
using System;
using System.Windows.Controls;
using System.Windows.Media;
using System.Threading;
using System.Windows.Threading;
namespace WpfGUIThreads
{
    public class RandomLetters
    {   // for random letters
        private static Random generator = new Random();
        private bool suspended = false; // true if thread is suspended
        private Label output; // Label to display output
        private string threadName; // name of the current thread

        // RandomLetters constructor
        public RandomLetters(Label label)
        {
            output = label;
        } // end
```

Using namespace **System.Windows.Media** for drawing capacities

`private` instance variables

```
// place random characters in GUI
public void GenerateRandomCharacters()
{
    // get name of executing thread
    threadName = Thread.CurrentThread.Name;

    while (true) // infinite loop; will be terminated from outside
    {
        // sleep for up to 1 second
        Thread.Sleep(generator.Next(1001));

        lock (this) // obtain lock
        {
            while (suspended) // loop until not suspended
            {
                Monitor.Wait(this); // suspend thread execution
            } // end while
        } // end lock

        // select random uppercase letter
        char displayChar = (char)(generator.Next(26) + 65);



        // display character on corresponding Label
        output.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
               new Action<char>((f) =>
                         { output.Content = threadName + ": " + f; }),
               displayChar);
    } // end while
} // end method GenerateRandomCharacters
```

Thread will sleep from 0-1 second

Infinite loop

Pick the next random character

Output to GUI

Store the current executing thread's name

Display the newly picked character to the GUI

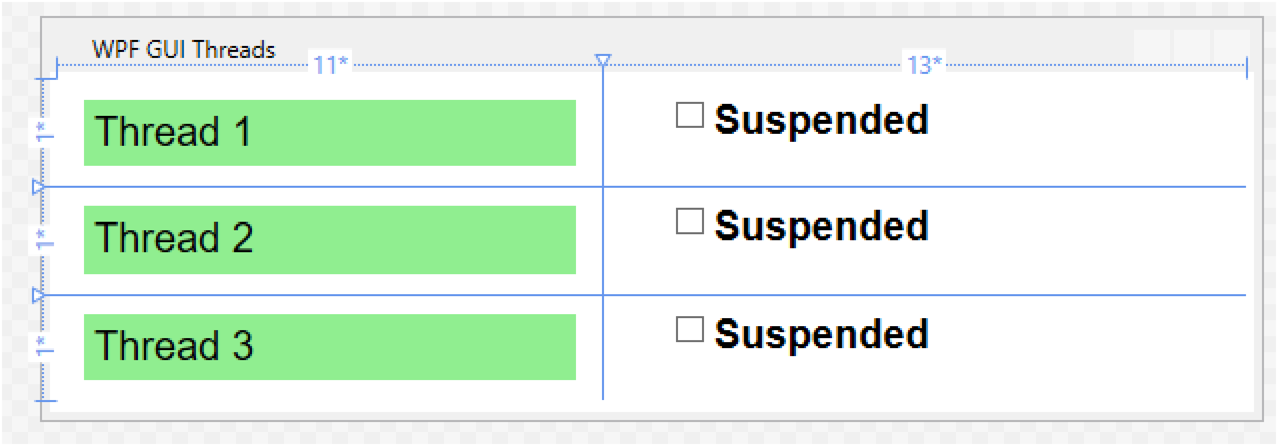
The **Action** delegate executes on the GUI thread

```
      // change the suspended/running state
      public void Toggle()
      {
         suspended = !suspended; // toggle bool controlling state

         // change label color on suspend/resume
         output.Background = suspended ? // set the Background
                                  Brushes.Red : Brushes.LightGreen;
         lock (this) // obtain lock
         {
            if (!suspended) // if thread re
               Monitor.Pulse(this);
         } // end lock
      } // end method Toggle
   } // end class RandomLetters
}
```

Toggle suspended

Change the background color depending if it is suspended

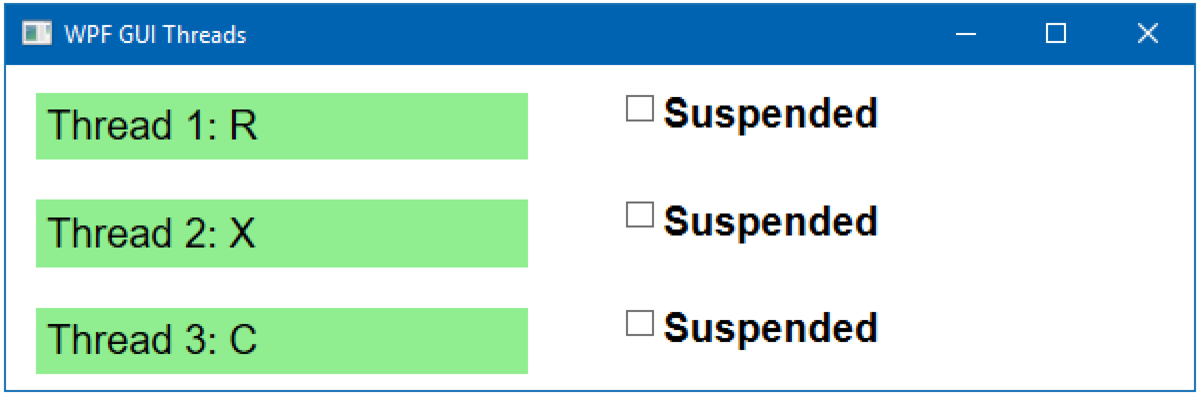
Allow the thread to continue

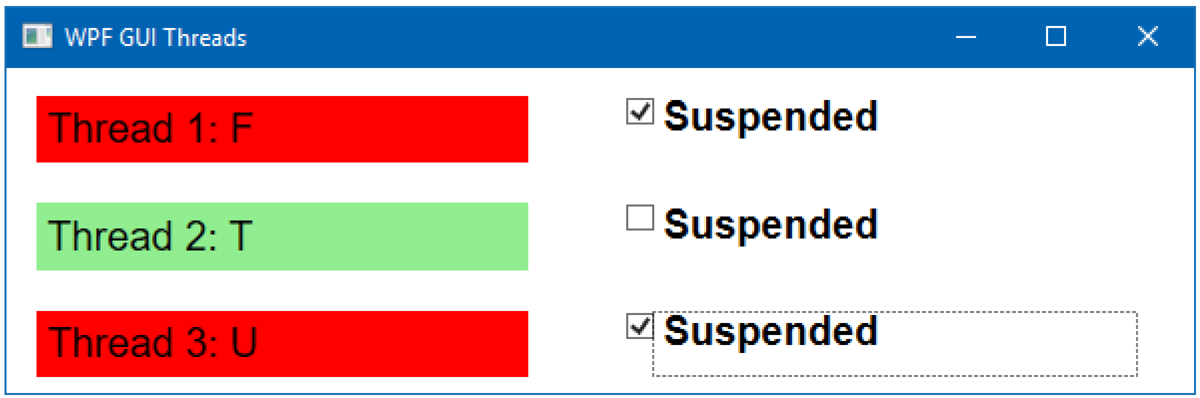

WPF GUI Threads
11*
13*
1*
1*
1*
Thread 1
Suspended
Thread 2
Suspended
Thread 3
Suspended

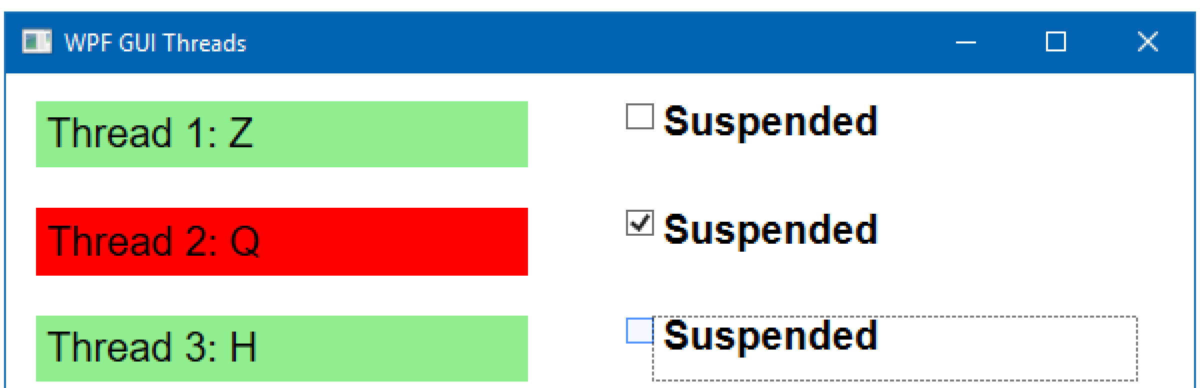
```csharp
using System.Windows;
using System.Threading;
namespace WpfGUIThreads
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private RandomLetters letter1; // first RandomLetters object
        private RandomLetters letter2; // second RandomLetters object
        private RandomLetters letter3; // third RandomLetters object
        public MainWindow()
        {
            InitializeComponent();
        }
        private void Window_Closing(object sender,
                                   System.ComponentModel.CancelEventArgs e)
        {   // close all running threads
            System.Environment.Exit(0);
        }// end method
```

```
        // suspend or resume the corresponding thread
       private void threadCheckBox_Checked(object sender, RoutedEventArgs e)
      {  // runs on Checked and Unchecked event
         if (sender == thread1CheckBox)
            letter1.Toggle();
         else if (sender == thread2CheckBox)
            letter2.Toggle();
         else if (sender == thread3CheckBox)
            letter3.Toggle();
      }// end method threadCheckBox_CheckedChanged

      private void Window_Loaded(object sender, RoutedEventArgs e)
      {
         // create first thread
         letter1 = new RandomLetters(thread1Label);
         Thread firstThread = new Thread(letter1.GenerateRandomCharacters);
         firstThread.Name = "Thread 1";
         firstThread.Start();

         // create second thread
         letter2 = new RandomLetters(thread2Label);
         Thread secondThread = new Thread(letter2.GenerateRandomCharacters);
         secondThread.Name = "Thread 2";
         secondThread.Start();

         // create third thread
         letter3 = new RandomLetters(thread3Label);
         Thread thirdThread = new Thread(letter3.GenerateRandomCharacters);
         thirdThread.Name = "Thread 3";
         thirdThread.Start();
      }
   }
}
```

Create and start the 3 threads


WPF GUI Threads
Thread 1: R
Suspended
Thread 2: X
Suspended
Thread 3: C
Suspended


WPF GUI Threads
Thread 1: F
Suspended
Thread 2: T
Suspended
Thread 3: U
Suspended


WPF GUI Threads
Thread 1: Z
Suspended
Thread 2: Q
Suspended
Thread 3: H
Suspended

15.13 Using Tasks

Queuing a work item to a thread pool can be useful, but it has its shortcomings. There **is no built-in way to know when the operation has finished** and **what the return value is**.

This is why the .NET Framework introduces the concept of a **Task**, which is **an object that represents some work that should be done**. The **Task** can tell you if the work is completed and if the operation returns a result, the **Task** gives you the result. A **task scheduler** is responsible for starting the **Task** and managing it. By default, the ***Task* scheduler uses threads from the thread pool** to execute the **Task** .

15.13 Using Tasks

Tasks can be used to make your application more responsive. If the thread that manages the user interface offloads work to another thread from the thread pool, **it can keep processing user events and ensure that the application can still be used**. But it doesn't help with scalability. If a thread receives a web request and it would start a new **Task**, it would just consume another thread from the thread pool while the original thread waits for results.

Executing a *Task* on another thread makes sense only if you want to keep the user interface thread free for other work or **if you want to parallelize your work on to multiple processors**.

15.13 Using Tasks

```
using System;
using System.Threading.Tasks;

namespace SampleTasks
{
    public static class Program
    {
        public static void Main()
        {   // Start a task in a worker thread
            Task t = Task.Run(() =>
            {
                for (int x = 0; x < 100; x++)
                {
                    Console.Write('*');
                }
            });
            t.Wait();// Join Task with main thread, waits Task to complete
            Console.WriteLine("\nEnd of task");
        }
    }
}
```

15.13 Using Tasks

`Task` provides the following powerful **features over thread and threadpool**.

1. Task allows you to **return a result**.
2. It gives better source code **control over running and waiting for a task**.
3. It **reduces the switching time among multiple threads**
4. It gives the ability to **chain multiple tasks together** and it can execute each task one after the other by using **`ContinueWith`**().
5. It **can create a parent/child relationship** when one task is started from another task.
6. **Task can cancel its execution** by using cancellation tokens.
7. Task leaves the CLR from the overhead of creating more threads; instead it **implicitly uses the thread from threadpool**.
8. Asynchronous implementation is easy in task, by using "**`async`**" and "**`await`**" keywords.
9. **`Task`** can wait for all of the provided **`Task`** objects to complete executions.

15.13 Using Tasks

The .NET Framework also has the **Task<T>** class that you can use if a **Task** should return a value. Use property **Result** of class **Task<T>.**

```
 public static void Main()
        {   // Start a task returning value in a worker thread
            Task<int> t = Task.Run(() =>
            {// The type of Task<int> is inferred
                by the Lambda return type
                return 42;// no need to write Task.Run<int>()
            });
            Console.WriteLine(t.Result); // Displays 42
        }
// alternatively
Task<TResult> mytask = new Task<TResult>(funcMethod);
myTask.Start();
```

where **funcMethod** is a method that has a **return type** of **TResult** type and **takes no input parameter**; in other words, there is a “**Func<TResult>**” delegate in the parameter of **Task** constructor.

15.13 Using Tasks

- ✓ **`Task<TResult>`** tells the **`Task`** operation to return an **`TResult`** value.
- ✓ **`myTask.Result;`** is a property that **returns a value** when the task gets completed and **blocks the execution of a calling thread** (in this case, its main thread) **until the task finishes its execution**.

15.13 Using Tasks

Methods & Properties	Explanation
Run()	Returns a Task that queues the work to run on ThreadPool
Start()	Starts a Task
Wait()	Wait for the specified task to complete its execution
WaitAll()	Wait for all provided task objects to complete execution
WaitAny()	Wait for any provided task objects to complete execution
ContinueWith()	Create a chain of tasks that run one after another
Status	Get the status of current task
IsCanceled	Get a bool value to determine if a task is canceled
IsCompleted	Get a bool value to determine if a task is completed
IsFaulted	Gets if the Task is completed due to an unhandled exception.
Factory	Provide factory method to create and configure a Task

Software Engineering Observation

You **should not re-use a task**. When you create and start a new task, the associated task scheduler makes sure that it's eventually executed. It doesn't necessarily begin executing immediately. Once the task has been removed from the queue of waiting tasks, which usually happens as cores become available, and run it is eventually finalized or disposed. Therefore you get the **`ObjectDisposedException`** if you attempt to reuse it.

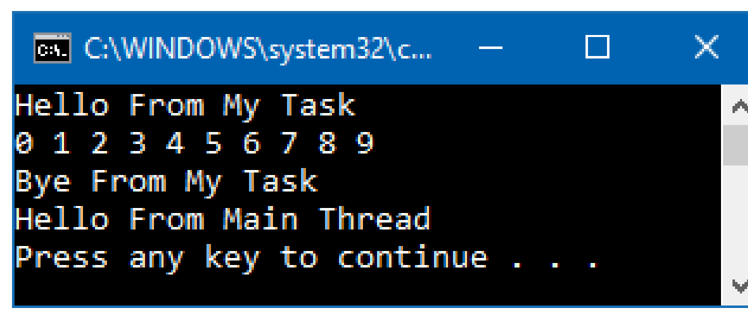
```
var task = new Task(DoSomeWork);
task.Start();
task.Wait();
task.Start(); // This throws an ObjectDisposedException!
```

15.13 Using Tasks

You can create and start a **Task** by using

Task.Factory.StartNew(Action<Void>)

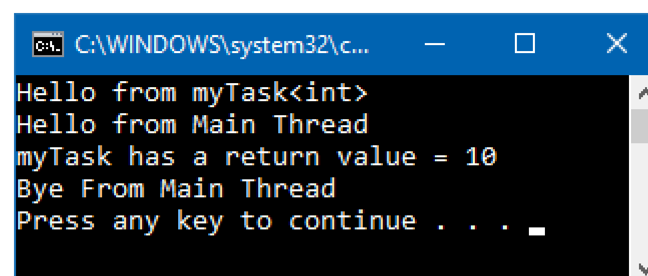
```
public static void Main() {
            //initialize and Start mytask and assign
            //a unit of work in the body of lambda exp
            Task mytask = Task.Factory.StartNew(new Action(MyMethod));
            mytask.Wait(); //Wait until mytask finish its job
                           //It's the part of Main Method
            Console.WriteLine("Hello From Main Thread");
 }
 private static void MyMethod()
 {
            Console.WriteLine("Hello From My Task");
            for (int i = 0; i < 10; i++)
            {
                Console.Write("{0} ", i);
            }
            Console.WriteLine();
            Console.WriteLine("Bye From My Task");
}
```



15.13 Using Tasks

You can create and start a **Task** by using
Task.Factory.StartNew<TResult>(Func<TResult>)

```
public static void Main()
        {
            Task<int> myTask = Task.Factory.StartNew<int>(FuncMethod);
            Console.WriteLine("Hello from Main Thread");
            //Wait the main thread until myTask is finished
            //and returns the value from myTask operation (myMethod)
            int i = myTask.Result;
            Console.WriteLine("myTask has a return value = {0}", i);
            Console.WriteLine("Bye From Main Thread");
        }
        private static int FuncMethod()
        {
            Console.WriteLine("Hello from myTask<int>");
            Thread.Sleep(1000);
            return 10;
        }
```



15.13 Using Tasks

The **`Task.Run`** method in no way obsoletes **`Task.Factory.StartNew`**, but rather should simply be thought of as a quick way to use **`Task.Factory.StartNew`** without needing to specify a bunch of parameters. **The Task.Run method is recommended to use when you don't need to have much fine-grained control over thread scheduling and its intricacies**.

In fact, **`Task.Run`** is actually implemented in terms of the same logic used for **`Task.Factory.StartNew`**, just passing in some default parameters. **`Task.Factory.StartNew`**() gives you the opportunity to define a lot of useful things about the thread you want to create, while **`Task.Run`** doesn't provide this.

For instance, lets say that you want **to create a long running task thread**. If a thread of the thread pool is going to be used for this task, then **this could be considered an abuse of the thread pool**. One thing you could do in order to avoid this would be to run the task in a separate thread. A **newly created thread that would be dedicated to this task** and would be destroyed once your task would have been completed. You cannot achieve this with the **`Task.Run`**, while you can do so with the **`Task.Factory.StartNew`**, like below:

```
Task.Factory.StartNew(...,
                      TaskCreationOptions.LongRunning);
```

Tip

`Task.Factory.StartNew()` saves performance cost when creating and starting a task compared to **`Task(…).Start()`** because **`Task(…).Start()`** **consumes more performance cost for creating and starting** a task.

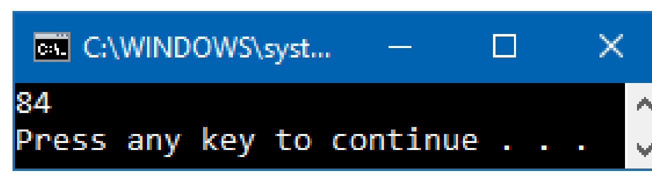
In .NET 4.5 and later editions, it is preferable to use **`Task.Run`** because it manages **`Task`** more efficiently than **`Task.Factory.StartNew`**.

`Task.Run()` returns and runs a task by assigning a unit of work in the form of a method ("**`myMethod`**").

15.13 Using Tasks

Because of the object-oriented nature of the **`Task`** object, one thing you can do is add ***a continuation task***. This means that you want another operation to execute as soon as the **`Task`** finishes. **`Task.Run<int>()`** takes a **`Func<int>`** delegate to reference a method that returns an integer value. This method gets executed by a task and a value gets returned by using the **`Result`** **property**.

```
public static void Main()
        {
            var  t = Task.Run(() =>
            { // runs a Func<int>() with zero input parameters
                return 42;
            }).ContinueWith((i) =>
            {
                return i.Result * 2;
            });
            Console.WriteLine(t.Result); // Displays 84
        }
```



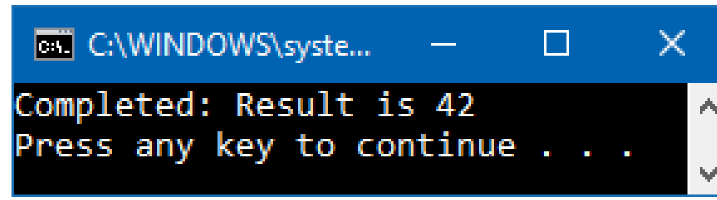
Note: the compiler sets the type of variable **`t`** to **`Task<int>`** by inference

15.13 Using Tasks

.NET offers conditional ***continuation of a task***. The **`ContinueWith`** method has a couple of overloads that you can use to configure when the continuation will run. This way you can add different continuation methods that will **run when an exception happens**, the **`Task`** is **canceled**, or the **`Task`** **completes successfully**.

15.13 Using Tasks

```
public static void Main()
      {
          Task<int> t = Task.Run(() =>
          {  // throw new Exception(); executes OnlyOnFaulted
              return 42;
          });
          t.ContinueWith((i) =>
          {
              Console.WriteLine("Canceled");
          }, TaskContinuationOptions.OnlyOnCanceled);
          t.ContinueWith((i) =>
          {
              Console.WriteLine("Faulted");
          }, TaskContinuationOptions.OnlyOnFaulted);
          var completedTask = t.ContinueWith((i) =>
          {
              Console.WriteLine("Completed: Result is {0}", i.Result);
          }, TaskContinuationOptions.OnlyOnRanToCompletion);
          completedTask.Wait();
      }
```



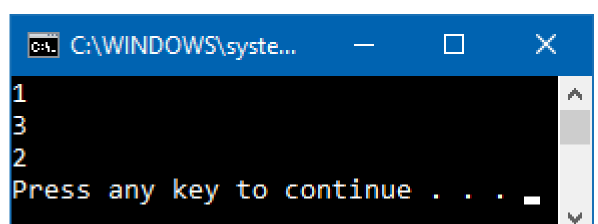
15.13 Using Tasks

Next to calling **`Wait`** on a single **`Task`**, you can also use the method **`WaitAll`** to wait for multiple *Tasks* to finish before continuing execution.

```
 public static void Main()
       {// may use also Task.WaitAll(tasks, 1000) to wait all until all tasks end but not more than 1000 ms
           Task[] tasks = new Task[3];
           tasks[0] = Task.Run(() =>
           {
               Thread.Sleep(1000);
               Console.WriteLine("1");
               return 1;
           });
           tasks[1] = Task.Run(() =>
           {
               Thread.Sleep(1000);
               Console.WriteLine("2");
               return 2;
           });
           tasks[2] = Task.Run(() =>
           {
               Thread.Sleep(1000);
               Console.WriteLine("3");
               return 3;
           }
           );
           Task.WaitAll(tasks);
or
       }
```

In this case, **all three Tasks are executed simultaneously**, and **the whole run takes approximately 1000ms instead of 3000ms**.

Instead of waiting until all tasks are finished, you can also wait until one of the tasks is finished. You use the **`WaitAny`** method for this.



15.13 Using Tasks

Improper usage of synchronization can easily lead to a deadlock, illustrated with the following example.

```
class Program
    {   // Sample deadlock with tasks
        //used as lock objects
        private static readonly object thislockA = new object();
        private static readonly object thislockB = new object();
        static void Main()
        {
            Task tsk1 = Task.Run(() =>
            {
                lock (thislockA)
                {
                    Console.WriteLine("thislockA of tsk1");
                    lock (thislockB)
                    {
                        Console.WriteLine("thislockB of tsk2");
                        Thread.Sleep(100);
                    }
                }
            }); // cont'd on the following slide
```

15.13 Using Tasks

```
            Task tsk2 = Task.Run(() =>
            {
                lock (thislockB)
                {
                    Console.WriteLine("thislockB of tsk2");
                    lock (thislockA)
                    {
                        Console.WriteLine("thislockA of tsk2");
                        Thread.Sleep(100);
                    }
                }
            });
            Task[] allTasks = { tsk1, tsk2 };
            Task.WaitAll(allTasks); // Wait for all tasks
            Console.WriteLine("Program executed successfully");
        }
    }
```

Here is how the application got frozen.

1. **`Tsk1`** acquires lock "**`thislockA`**".

2. **`Tsk2`** acquires lock "**`thislockB`**".

3. **`Tsk1`** attempts to acquire lock "**`thislockB`**", but it is already held by Tsk2 and thus Tsk1 blocks until "**`thislockB`**" is released.

4. Tsk2 attempts to acquire lock "**`thislockA`**", but it is held by **`Tsk1`** and thus **`Tsk2`** blocks until "**`thislockA`**" is **`released`**. At this point, **both threads are blocked and will never wake up**.

Hence, the application froze.

15.13 Using Tasks

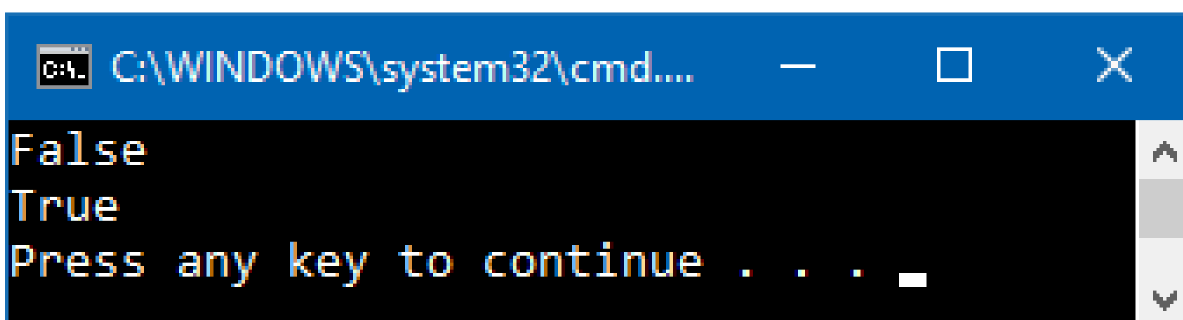
The following example shows the basic pattern for task cancellation. **Note that the token is passed to the user delegate and to the task instance itself.**

```
static void Main(string[] args)
        {
            CancellationTokenSource tokenSource = new CancellationTokenSource();
            var token = tokenSource.Token;
            Task taskWithToken = new Task(
                () =>
                {
                    while (true)
                    {
                        if (token.IsCancellationRequested)
                        {
                            break;
                        }
                    }
                }, token
            );
```

15.13 Using Tasks

```
    taskWithToken.Start();
    while (taskWithToken.Status != TaskStatus.Running)
    {
           //Wait until task is running
    }
    tokenSource.Cancel();  //cancel...
    //...and wait for the action within the task to complete
    taskWithToken.Wait();
    Console.WriteLine(taskWithToken.Status == TaskStatus.Canceled);
    Console.WriteLine(taskWithToken.Status == TaskStatus.RanToCompletion);
  }
```

Once a **Task** has a Status of "**Running**", calling **Cancel**() on the **CancellationTokenSource** no longer has an effect on the actual **Task**, it is up to the **Action** within the **Task** to respond to the token's message. The **Task** is cancelled while running, but it **RanToCompletion**.



15.11 Build asynchronous methods

Long-running **CPU-bound tasks** can be handed to another thread by using the **Task** object. But **when doing work that's input/output (I/O)–bound**, things go a little differently.

When your application is executing an **I/O operation** on the primary application thread, Windows notices that **your thread is waiting for the I/O operation to complete**. Maybe you are accessing some file on disk or over the network, and this could take some time.

Because of this, **Windows pauses your thread so that it doesn't use any CPU resources**. But while doing this, it still uses memory, and the thread can't be used to serve other requests, which in turn will lead to new threads being created if requests come in. **Instead of blocking your thread until the I/O operation finishes, you get back a Task object that represents the result of the asynchronous operation**. **By setting a continuation on this Task**, you can continue when the I/O is done. **In the meantime, your thread is available for other work**. When the I/O operation finishes, Windows notifies the runtime and the continuation **Task** is scheduled on the thread pool.

15.11 Build asynchronous methods

You use the `async` keyword to mark a method for asynchronous operations. This way, you signal to the compiler that something asynchronous is going to happen. The compiler responds to this by transforming your code into a ***state machine.*** A method marked with `async` just starts running synchronously on the current thread. What it does **is enable the method to be split into multiple pieces**. The boundaries of these pieces are marked with the `await` keyword.

When you use the `await` keyword, **the compiler generates code that will see whether your asynchronous operation is already finished**. If it is, your method just continues running synchronously. If it's not yet completed, **the state machine will hook up a continuation method that should run when the `Task` completes**. Your method yields control to the calling thread, and this thread can be used to do other work.

15.11 Build asynchronous methods

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

In the code sample 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 HTML 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. (I-O bound task!)**
- **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. (CPU bound task!)**
- **Finally, it prints out the number of characters that were found for the two web sites.**

15.11 Build asynchronous methods

```
class MyDownloadString
{
   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 );
   }
```

15.11 Build asynchronous methods

```
 private int CountCharacters( int id, string uriString )
   {// I/O- bound task
      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 )
   { // CPU- bound task
      for ( long i=0; i < value; i++ )     ;
      Console.WriteLine( " End counting {0} : {1, 4:N0} ms", id,
                                    sw.Elapsed.TotalMilliseconds );
   }
}
```

15.11 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

15.11 Build asynchronous methods

C#'s `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.**

15.11 Build asynchronous methods

`DoRun` can then continue on and make the four calls to CPU bound tasks like `CountToALargeNumber`, while the two calls to I-O bound tasks like `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. If a result isn’t ready yet, execution blocks and waits until it is.

15.11 Build asynchronous methods

```
using System.Threading.Tasks;
class MyDownloadString
{
   Stopwatch sw = new Stopwatch();
   public void DoRun()
   {
     const int LargeNumber = 6000000;
     sw.Start();
     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 );
   }
```

15.11 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;
}
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 );
}
}
```

15.11 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

15.11 Build asynchronous methods

The new version is 32 percent 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!

The next slide uses **`GetStringAsync`** of **`HttpClient`** to run asynchronous code internally and returns a **`Task<string`**> to the caller that will finish when the data is retrieved. In the meantime, your thread can do other work. The nice thing about **`async`** and **`await`** is that **they let the compiler do the thing it's best at: generate code in precise steps**. The **`await`** keyword enables you to **write code that looks synchronous but behaves in an asynchronous way**

15.11 Build asynchronous methods

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
namespace AsyncHttpClient
{
    public static class Program
    {
        public static void Main()
        {
            string result = DownloadContentAsync().Result;
            Console.WriteLine(result);
        }
        public static async Task<string> DownloadContentAsync()
        {  // use HttpClient instead of WebClient
            using (HttpClient client = new HttpClient())
            {   // Note: GetStringAsync() returns a Task<string>
                string result = await client.GetStringAsync("http://www.microsoft.com");
                return result;
            } // returns the HTML of http://www.microsoft.com
        }
    }
```

Software Engineering Observation 15.1

Doing a CPU-bound task is different from an I/O-bound task.

CPU-bound tasks always use some thread to execute their work.

An **asynchronous I/O-bound task** doesn't use a thread until the I/O is finished.

15.11a The Structure of the `async/await` Feature

In contrast to synchronous processing, an *asynchronous method returns to the calling method before it finishes all its* work. C#'s `async/await` feature allows you to create and use asynchronous methods. The feature comprises three components:

- **The *calling method is the method that calls an* `async` *method and then continues* on its way while the `async` method performs its tasks, either on the same thread or a different thread.**
- **The `async` method is the method that sets up its work to be done asynchronously, and then returns early to the calling method.**
- **The `await` expression is used inside the `async` method and specifies the task that needs to be performed asynchronously. An `async` method can contain any number of `await` expressions, although the compiler produces a warning message if there isn't at least one.**

15.11a The overall structure of the `async/await` feature

```
class Program
{
    static void Main()
    {
        ...
        Task<int> value = DoAsyncStuff.CalculateSumAsync(5, 6);
        ...
    }
}

static class DoAsyncStuff
{
    public static async Task<int> CalculateSumAsync(int i1, int i2)
    {
        int sum = await TaskEx.Run( () => GetSum( i1, i2 ) );
        return sum;
    }
    ...
}
```

Ca[redacted]ing Method

Async Method

Await Expression

15.11a *The overall structure of the* ***`async/await`*** *feature*

`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*.")

15.11a *The overall structure of the* `async/await` *feature*

`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)

15.11b The *async* method

An **async** method is a method that returns to the calling method before completing all its work, and then completes its work while the calling method continues its execution. Syntactically, an **async** method has the following characteristics:

- ✓ It has the **async** method modifier in the method header.
- ✓ It contains one or more **await** expressions. These expressions represent tasks that can be done asynchronously.
- ✓ It must have one of the following three return types. In the second and third cases that is, **Task** and **Task<T>**, the returned object represents a chunk of work that will be completed in the future, while both the calling method and the **async** method can continue processing.
  - **void**
  - **Task**
  - **Task<T>**

15.11b The *async* method

As of C# 7.0 `async` methods can return `ValueTask<T>`. Then if performance analysis proves it worthwhile should a [`ValueTask<TResult>`](#) be used instead of `Task<TResult>`.

```
public Task<int> GetIdAsync()
{
 return Task.FromResult (5);
}
```

This code is simple, does not create the whole async state machine magic but has one problem. It allocates as Task is class. Of course for the majority of the cases this won't be a problem. If you want to write a really fast code though, you might consider using an alternative called *ValueTask*, that is a struct. For the cases, where the implementation of a method might be synchronous, you could consider this as a better signature.

```
public ValueTask<int> GetIdAsync()
{
 return new ValueTask(5);
}
```

15.11b The `async` method

As of C# 7.0 `async` methods can return `ValueTask<T>`. Then if performance analysis proves it worthwhile should a `ValueTask<TResult>` be used instead of `Task<TResult>`.

- An `async` method can `have any number of formal parameters` of `any types`. None of the parameters, however, can be `out` or `ref` parameters.
- By convention, the name of an `async` method should end with the suffix `Async`.
- Besides methods also lambda expressions and anonymous methods can act as `async` objects.

15.11b The *async* method

The **return** type must be one of the three following types. Notice that two of the return types include the **Task** class.

- **Task<T>**: 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**: 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();
```

15.11b The `async` method

`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.

Examples for these three return types are shown in the following slides.

15.11b The *async* method

```
using System;
using System.Threading.Tasks;

class Program
{
    static void Main() {
        Task<int> value = DoAsyncStuff.CalculateSumAsync( 5, 6 );
        // Do other processing ...
        Console.WriteLine( "Value: {0}", value.Result );
    }
}

static class DoAsyncStuff
{
    public static async Task<int> CalculateSumAsync(int i1, int i2)   {
        int sum = await Task.Run( () => GetSum( i1, i2 ) );
        return sum;
    }

    private static int GetSum( int i1, int i2 ) { return i1 + i2; }
}
```

15.11b The *async* method

```
using System;
using System.Threading.Tasks;

class Program
{
    static void Main() {
        Task someTask = DoAsyncStuff.CalculateSumAsync(5, 6);
        // Do other processing
        someTask.Wait();
        Console.WriteLine( "Async stuff is done" );
    }
}

static class DoAsyncStuff
{
    public static async Task CalculateSumAsync( int i1, int i2 ) {
        int value = await Task.Run( () => GetSum( i1, i2 ) );
        Console.WriteLine("Value: {0}", value );
    }

    private static int GetSum( int i1, int i2 ) { return i1 + i2; }
}
```

15.11b The *async* method

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
   static void Main() {
      DoAsyncStuff.CalculateSumAsync(5, 6);
      // Do other processing
      Thread.Sleep( 200 );
      Console.WriteLine( "Program Exiting" );
   }
}

static class DoAsyncStuff
{
   public static async void CalculateSumAsync(int i1, int i2) {
      int value = await Task.Run( () => GetSum( i1, i2 ) );
      Console.WriteLine( "Value: {0}", value );
   }

   private static int GetSum(int i1, int i2) { return i1 + i2; }
}
```

15.11b The *async* method

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 following 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.

15.11b The *async* method

```
private async Task<int> CountCharactersAsync(
 string site )
{
   WebClient wc = new WebClient();
   string result = await
     wc.DownloadStringTaskAsync(new Uri( site ));
   return result.Length;
}
```

15.11c The await Expression

The `await` expression specifies a task to be done asynchronously. The syntax of the `await` expression is shown below and consists of the `await` keyword followed by an awaitable object, which is called the `task`. The `task` might or might not be an object of type `Task`. By default, this task is run asynchronously on the current thread.

```
await task
```

An awaitable object is an instance of an *awaitable type* like type `Task`.

15.11c The await Expression

With .NET 4.5, Microsoft released a large number of `new` and `reworked` asynchronous methods, that return objects of type `Task<T>`.

You can plug these right into your `await` expression, and they'll work asynchronously on your current thread.

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`, `WriteAsync`,`ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.
They use an implementation that makes use of actual asynchronous I/O. This way, they don't use a thread while they are waiting on the hard drive of your system to read or write some data.

15.11c The await Expression

The `WebClient.DownloadStringTaskAsync` method is one example of these asynchronous methods. The following code is an example of its usage:

```
Uri site = new
    Uri("http://www.microsoft.com" );
WebClient wc = new WebClient();
string result = await
          wc.DownloadStringTaskAsync( site );
```

15.11c The await Expression

```
 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`

15.11c The await Expression

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<>` (for returning a `Task<>`) or (`Task.CompletedTask` for returning just a `Task`) is more efficient than an `async` method returning a value. In general, synchronous methods should be written and called in the traditional way.

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

15.11c The await Expression

```
// 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();
```

15.11c The await Expression

Although there are now a number of BCL methods that return objects of type `Task<T>`, you'll most likely have your own methods that you want to use as the task for an `await` expression. The easiest way to do that is to create a `Task` from your method using the `Task.Run` method. One very important fact about the `Task.Run` method is that *it runs your method on a different thread.*

One signature of the `Task.Run` method is the following, which takes a `Func<TReturn>` delegate as a parameter. You'll remember `Func<TReturn>` is a predefined delegate that takes no parameters and returns a value of type `TReturn`:

```
Task.Run( Func<TReturn> func )
```

15.11c The await Expression

So, to pass your method to the `Task.Run` method, you need to create a delegate from it. The following code shows three ways to do this. In the code, method `Get10` has a form compatible with a `Func<int>` delegate since it takes no parameters and returns an `int`.

- In the first instance, which is in the first two lines of method `DoWorkAsync`, a `Func<int>` delegate named `ten` is created using `Get10`. That delegate is then used in the `Task.Run` method in the next line.

15.11c The await Expression

- In the second instance, a `Func<int>` delegate is created right in the `Task.Run` method's parameter list.
- The last instance doesn't use the `Get10` method at all. It uses the `return` statement that comprises the body of the `Get10` method, and uses it as the body of a lambda expression compatible with a `Func<int>` delegate. The lambda expression is implicitly converted to the delegate.

15.11c The await Expression

```
class MyClass
{
   public int Get10()                              // Func<int> compatible
   {
      return 10;
   }

   public async Task DoWorkAsync()
   {
      Func<int> ten = new Func<int>(Get10);
      int a = await Task.Run(ten);

      int b = await Task.Run(new Func<int>(Get10));

      int c = await Task.Run(() => { return 10; });

      Console.WriteLine("{0}  {1}  {2}", a, b, c);
   }

class Program
{
   static void Main()
   {
      Task t = (new MyClass()).DoWorkAsync();
      t.Wait();
   }
}
```

```
10  10  10
```

15.11c The await Expression

We used the signature for `Task.Run` that takes a `Func<TResult>` as the parameter. There are a total of eight overloads for the method, which are shown in the following Table.

15.11c The await Expression

Return Type	Signature
Task	Run(Action action)
Task	Run(Action action, CancellationToken token)
Task<TResult>	Run(Func<TResult> function)
Task<TResult>	Run(Func<TResult> function, CancellationToken token)
Task	Run(Func<Task> function)
Task	Run(Func<Task> function, CancellationToken token)
Task<TResult>	Run(Func<Task<TResult>> function)
Task<TResult>	Run(Func<Task<TResult>> function, CancellationToken token)

The Types of Delegates That Can Be Sent to a `Task.Run` Method As the First Parameter

Delegate Type	Delegate Signature	Meaning
`Action`	`void Action()`	A method that takes no parameters and returns no value
`Func<TResult>`	`TResult Func()`	A method that takes no parameters and returns an object of type T
`Func<Task>`	`Task Func()`	A method that takes no parameters and returns a simple `Task` object
`Func<Task<TResult>>`	`Task<TResult> Func()`	A method that takes no parameters and returns an object of type `Task<T>`

Four await statements that use the `Task.Run` method to run methods with the four different delegate types

```
static class MyClass
{
   public static async Task DoWorkAsync()
   {                                       Action
                                             ↓
      await Task.Run(() => Console.WriteLine(5.ToString()));
                                               TResult Func()
                                                  ↓
      Console.WriteLine((await Task.Run(() => 6)).ToString());
                                                     Task Func()
                                                        ↓
      await Task.Run(() => Task.Run(() => Console.WriteLine(7.ToString())));
                                              Task<TResult> Func()
                                                     ↓
      int value = await Task.Run(() => Task.Run(() => 8));
      Console.WriteLine(value.ToString());
   }
}

class Program
{
   static void Main()
   {
      Task t = MyClass.DoWorkAsync();
      t.Wait();
      Console.WriteLine("Press Enter key to exit");
      Console.Read();
   }
}
```

```
5
6
7
8
```

15.11c The await Expression

The `await` expression can be used anywhere any other expression can be used (as long as it's inside an async method). In the code above, the four `await` expressions are used in three different positions.

- The first and third instances use the `await` expression as a statement.
- In the second instance, the `await` expression is used as the parameter to the `WriteLine` method call.
- The fourth instance uses the `await` expression as the right-hand side of an assignment statement.

15.11c The await Expression

Suppose, however, that you have a method that doesn't match any of the four delegate forms. For example, suppose that you have a method named `GetSum` that takes two int values as input and returns the sum of the two values. This isn't compatible with any of the four acceptable delegates. To get around this, you can create a lambda function in the form of an acceptable `Func` delegate, whose sole action is to run the `GetSum` method, as shown in the following line of code:

```
int value = await Task.Run(() => GetSum(5, 6));
```

The lambda function `() => GetSum(5, 6)` satisfies the `Func<TResult>` delegate because it is a method that takes no parameters, but returns a single value.

15.11c The await Expression

```
static class MyClass
{
   private static int GetSum(int i1, int i2)
   {
      return i1 + i2;
   }

   public static async Task DoWorkAsync()
   {                                 TResult Func()
                                          ↓
      int value = await Task.Run( () => GetSum(5, 6) );
      Console.WriteLine(value.ToString());
   }
}

class Program
{
   static void Main()
   {
      Task t = MyClass.DoWorkAsync();
      t.Wait();
      Console.WriteLine("Press Enter key to exit");
      Console.Read();
   }
}
```

```
11
Press Enter key to exit
```

15.12 Making UI Responsive

In any .NET GUI Application (Windows Form, **`WPF`**, ASP.NET, etc.), the User Interface (UI) becomes unresponsive when **a complex and time-consuming operation is executed during an event**. **A UI** (user-interface) thread manages the life cycle of UI controls (buttons, textbox, etc.), and it is commonly used to handle user inputs and respond to user events.

In order to make UI responsive, it is essential to not execute complicated and time-consuming operations on a UI thread. Instead, these time-consuming operations must run on separate tasks, controlled by **`async`** and **`await`** keywords. Doing this, the UI thread becomes free and available to respond to any user input.

15.12 Making UI Responsive

One other thing that's important when working with asynchronous code is the concept of a **`SynchronizationContext`**, which connects its application model to its threading model. For example, a WPF application uses a single user interface thread and potentially multiple background threads to improve responsiveness and distribute work across multiple CPUs. An ASP.NET application, however, uses threads from the thread pool that are initialized with the correct data, such as current user and culture to serve incoming requests.

The **`SynchronizationContext`** abstracts the way these different applications work and makes sure that you end up on the right thread when you need to update something on the UI or process a web request.

15.12 Making UI Responsive

The **`await`** keyword makes sure that the current **`SynchronizationContext`** is **saved and restored** when the task finishes.

When using **`await`** inside a WPF application, this means that after your **`Task`** finishes, your program continues running on the user UI thread.

It's important to figure out which commands are taking a longer time. Put those commands on a separate method whose **return type** is **`Task/Task<T>`**.

Next, use the "**`async`**" keyword on its signature and write the "**`await`**" keyword before the method of type "**`Task`**" is called.

15.12 Making UI Responsive

```
private async void Button_Click(object sender, RoutedEventArgs e)
        {
            using (HttpClient client = new HttpClient())
            {
                string content = await httpClient.GetStringAsync(
                                     "http://www.microsoft.com");
                outputLabel.Content = content;
            }
        }
```

Note that the current **SynchronizationContext** is **saved and restored** when the task finishes.

Software Engineering Observation 15.1

When using `async` and `await` keep in mind that you should never have a method marked `async` without any `await` statements.

You should also avoid returning `void` from an `async` method except when it's an event handler.

15.12 Making UI Responsive

The following steps are essential to **execute any method in a GUI asynchronously**:

1. The return type of an event-method doesn't change. It is always **`void`**. But the return type of a normal method must change to **`Task<return_type>.`**

2. You must **add** the "**`async`**" keyword infront the **`return_type/Task<return_type>`** of any method.

3. You must use the "**`await`**" keyword when a method is called whose **return** type is "**`Task/Task<T>`**".

15.12 Making UI Responsive

In any normal method, it's important to figure out which operations are time-consuming and execute them on a separate task. Use **`async`** and **`await`** on a normal method

```
private async void Button_Click(object sender, RoutedEventArgs e)
        {
            outputLabel.Content = “Hello World”;
            await NormalMethodAsync();
            outputLabel.Content = “Work finished”;
         }
private async Task NormalMethodAsync()
// note, method returns Task not void
            {
               await DoComplicatedTask();
            }
```

15.12 Making UI Responsive

In another version you may use **`Task<T>`** of Lambda expression in an event handler.

```
private async void Button_Click(object sender, RoutedEventArgs e)
        {
            outputLabel.Content = “Hello World”;
            string content = await NormalMethodAsync();
            outputLabel.Content = “Work “ + content;
         }
private async Task<string> NormalMethodAsync()
            {
              Task<string> task = Task.Run(() =>”finished”);
              return task;
            }
```

15.12 Making UI Responsive

```
private async void Button_Click(object sender, RoutedEventArgs e)
        {
          outputLabel.Content = “Hello World”;
          Func<Task> asyncLambda=(async()=>await { MyWait(5000);} );
          await asyncLambda();// asynchronous Lambda
          outputLabel.Content = “Work completed.“;
         }
private Task MyWait( int millisec)
         {
           Task task = Task.Run(() =>Thread.Sleep(millisec);
           return task;
          }
```

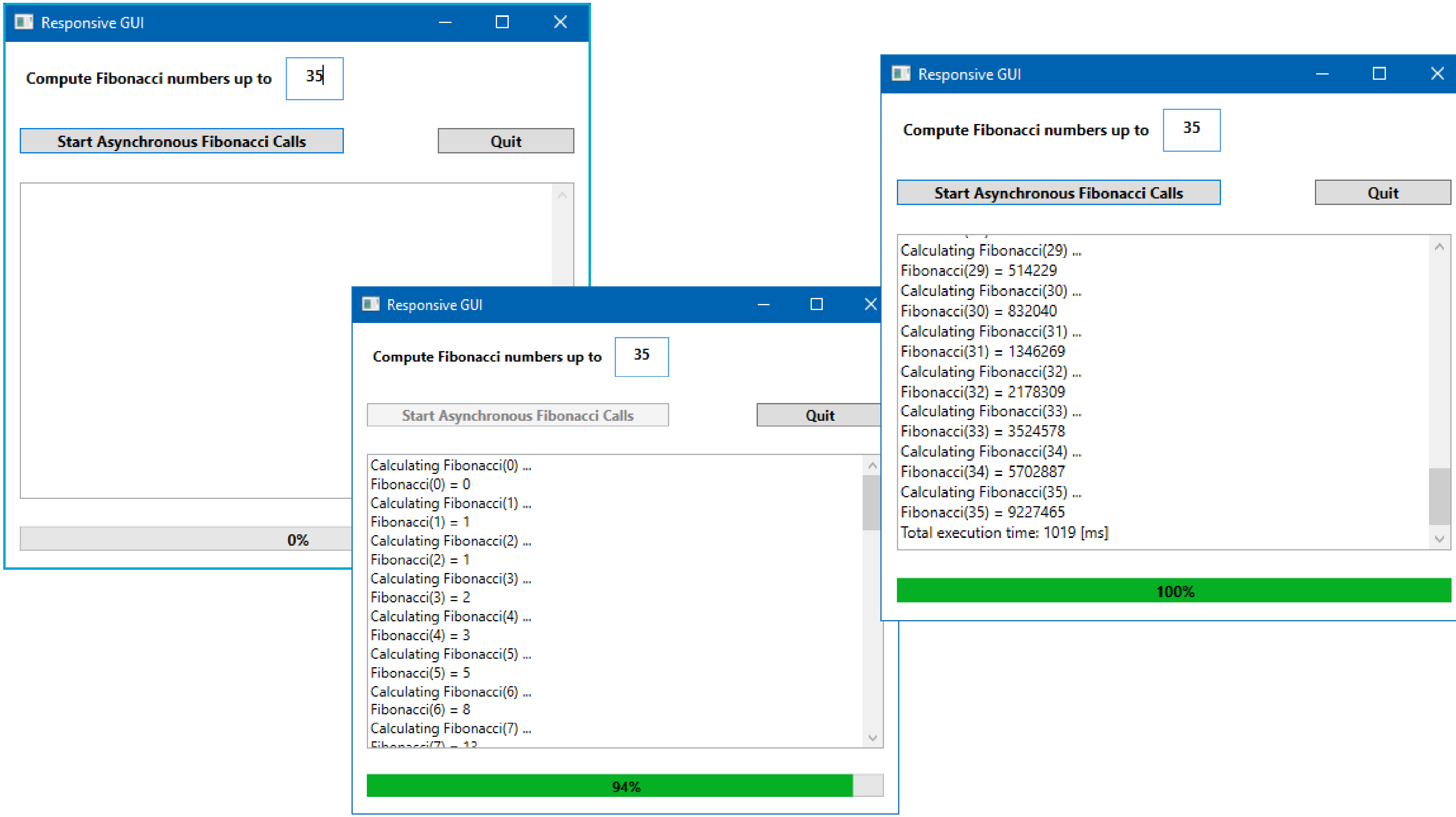
15.12.1 Asynchronous Task in WPF

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

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n – 1) + 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 We launch the calculation *asynchronously and have it execute on a separate thread so the GUI* remains *responsive*.

15.12.1 Asynchronous Task in WPF



15.12.1 Asynchronous Task in WPF

The **`Start Asynchronous Fibonacci Calls`** button's event handler (**`StartButton_Click()`**) reads the **`number`** in the **`TextBox`** and **awaits** the execution of an asynchronous **`Task`** (**`ComputeFibonacciNumbersAsync)`** to **display all Fibonacci numbers that are less or equal to the input number**. Note, **`async`** **does not run your code on a thread pool thread and the execution remains on the UI thread, while awaiting the Task to complete.**

The event handler is declared **`async`** 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.**

15.12.1 Asynchronous Task in WPF

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // initialize variables

    int maxFibNumber = int.Parse(txtInput.Text);
    if (maxFibNumber < 1)
    {
        outputTextBox.Text += "Wrong input. Enter a number greater than 0.";
        return;
    }
    startButton.IsEnabled = false;
    pbStatus.Value = pbStatus.Minimum;
    pbStatus.Maximum = 100;
    // await the execution of the asynchronous Task
    await ComputeFibonacciNumbersAsync(maxFibNumber);
    // the Task ran to completion
    startButton.IsEnabled = true;
    sw.Reset();
}
```

15.12.1 Asynchronous Task in WPF

`ComputeFibonacciNumbersAsync` wraps in a `Task` (namespace `System.Threading.Tasks`) the results of the execution of method Fibonacci() .

Note that here `ComputeFibonacciNumbersAsync` runs on the UI thread. Because the computation of the Fibonacci numbers is CPU- bound task we use `Task.Run<>()` in order to create in a loop multiple Tasks that run on TPL threads and thus make the GUI responsive. It is not wise to use `Task.FromResult<>()` as it would block the UI thread for large numbers of Fibonacci, because this Task would run on the UI thread.

Here each of these threads returns control to the UI thread and therefore we can execute directly commands on the UI thread (write text in the `TextArea` and the change the property `Value` of the `ProgressBar`). When any of the `Task.Run<>()` is already complete, execution continues with the following command that displays the computation result in the `outputTextBox`. . Otherwise, control returns to `UI thread` until the computation result becomes available. This allows the GUI to remain *responsive while* the Task executes. Once `ComputeFibonacciNumbersAsync` completes, `program` displays the `Total` execution time.

15.12.1 Asynchronous Task in WPF

```
private async Task ComputeFibonacciNumbersAsync(int maximum)
{
    long fn = 0;
    sw.Start();
    //Start computing the Fibonacci numbers
    for (int i = 0; i <= maximum; i++)
    {
        long k;
        k = (long)i;
        // BeginInvoke is async unlike Invoke
        outputTextBox.Text += String.Format("Calculating Fibonacci({0}) ...\r\n", i);

        fn = await Task.Run<long>(()=>Fibonacci(k));// wrap in a Task the execution of Fibonacci()

        outputTextBox.Text += String.Format("Fibonacci({0}) = {1}\r\n", i, fn);
        pbStatus.Value = (i * 100) / maximum;//Note:pbStatusTextBlock is bound to pbStatus.Value
    }
    outputTextBox.Text += String.Format("Total execution time: {0} [ms] \r\n", sw.ElapsedMilliseconds);
    sw.Reset();
}
```

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

15.12.1 Asynchronous Task in WPF

The version of `class Task`*'s static method* `Run` **receives a** `Func<TResult>` **delegate as an argument and executes a method in a** *separate TPL 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 we use the** *lambda expression*

```
()=>Fibonacci(k)
```

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

15.12.1 Asynchronous Task in WPF

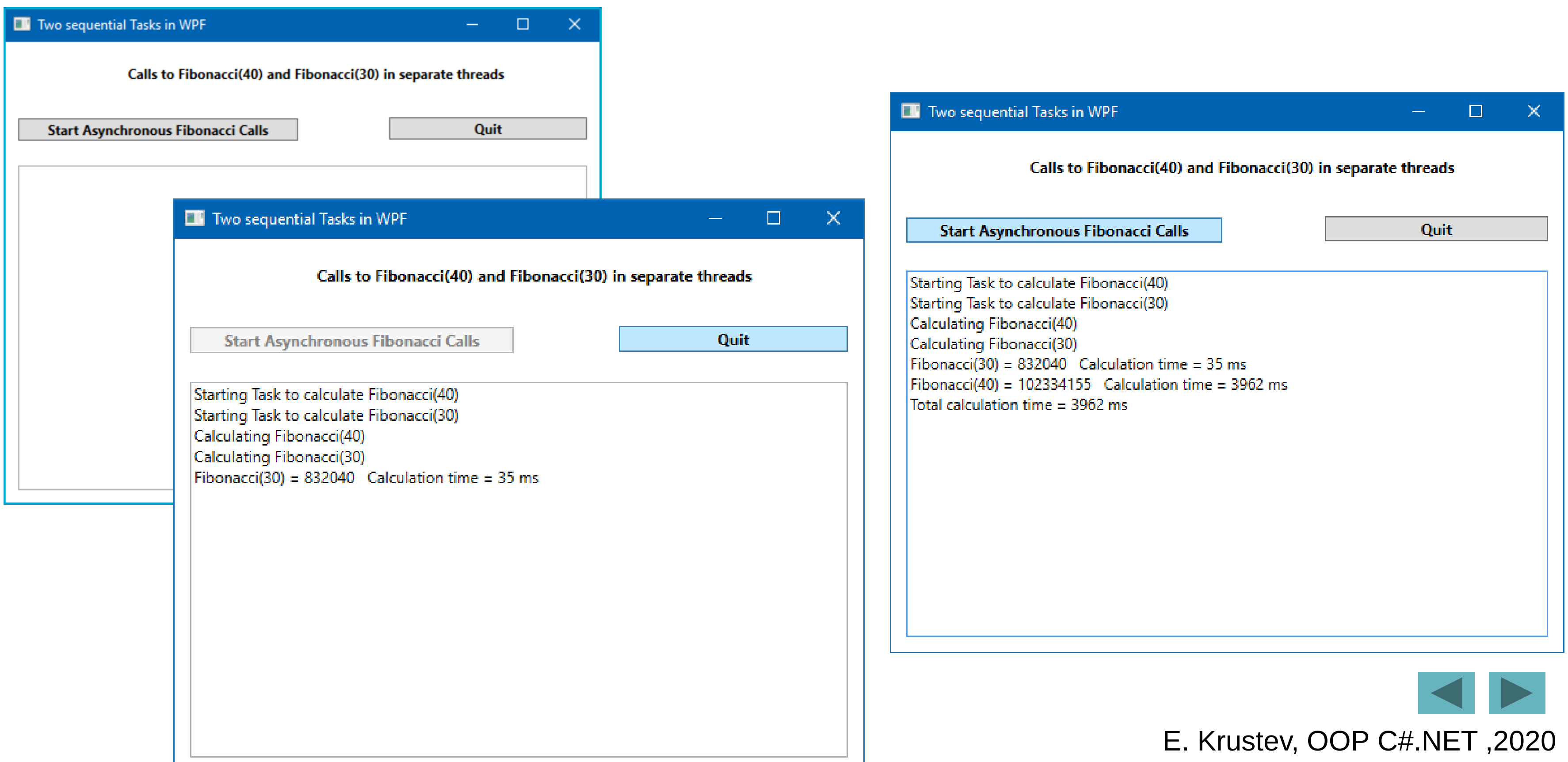
Notice that the `Text` property of `TextBlock pbStatusTextBlock` is bound to the property `Value` of the `Progressbar pbStatus`. Once we update `pbStatus.Value` so is the property `pbStatusTextBlock`.Text.

```
<ProgressBar x:Name="pbStatus" Margin="10,12,10,8" Grid.Row="3" Minimum="0" Maximum="100"
             RenderTransformOrigin="0.502,-0.056"/>
<TextBlock x:Name="pbStatusTextBlock" Text="{Binding ElementName=pbStatus, Path=Value, StringFormat={}{0:0}%}"
           Grid.Row="3" Margin="207,14,205,10" RenderTransformOrigin="0.759,-0.375" FontWeight="Bold"
           TextAlignment="Center" />
```

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

15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

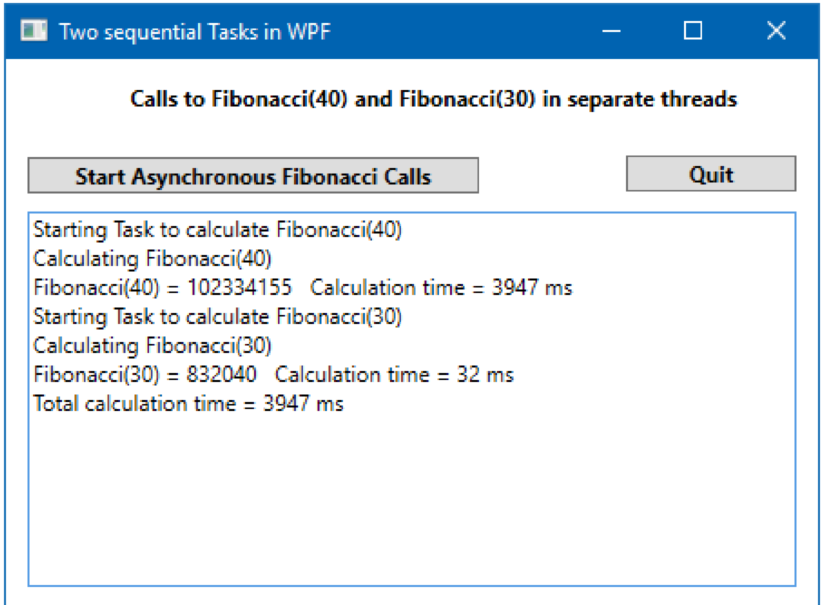
The following example uses the recursive Fibonacci method, but the two initial calls to Fibonacci execute in *separate Tasks* to compute the Fibonacci numbers 40 and 30, while keeping the GUI responsive.



15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

Here we apply a **sequential pattern** for **Task** execution by awaiting the task's execution one after another, waiting for one to finish before starting the next (**StartButton_ClickV1()**). The execution of each task is performed by the **async** method **StartFibonacciAsync**().

The first **Task** computes **Fibonacci(40)** and it is more CPU- intensive task than the second task that computes **Fibonacci** (30). However, because of the sequential pattern in their execution **Fibonacci(40)** outputs before **Fibonacci(30)** in the GUI.



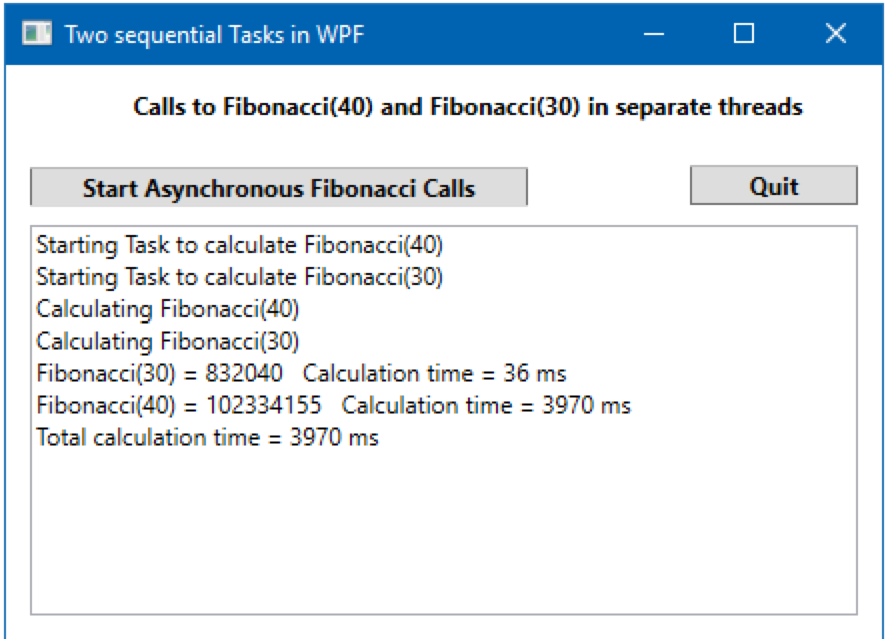
15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
private async void StartButton_ClickV1(object sender, RoutedEventArgs e)
{  // uses 2 async Tasks to compute Fibonacci
    startButton.IsEnabled = false;

    outputTextBox.Text = "Starting Task to calculate Fibonacci(40)\r\n";
    // await Task to perform Fibonacci(40) calculation
    long r1 = await StartFibonacciAsync(40);

    outputTextBox.AppendText("Starting Task to calculate Fibonacci(30)\r\n");
    // create Task to perform Fibonacci(30) calculation
    long r2 = await StartFibonacciAsync(30);
    // both tasks are ran to completion
    outputTextBox.AppendText(String.Format(
                    "Total calculation time = {0} ms\r\n",
                    Math.Max(r1, r2)));

    startButton.IsEnabled = true;
}
```

Awaits two Tasks to compute Fibonacci

15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

Another option to **execute both tasks in parallel (StartButton_ClickV2()),** **where Fibonacci(40**) and **Fibonacci** (30) run in dedicated new threads and **wait for both of them to complete**. This way, the **fastest Task** (in this case **Fibonacci** (30)) **returns first** its output to the GUI although it is started after **Fibonacci** (40) . The tasks **execute in threads different** than the **UI** thread, however**, on completion they return control to the UI- thread** and we can write directly **Text** to the **outputTextBox**.



15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
private async void StartButton_ClickV2(object sender, RoutedEventArgs e)
{  // uses 4 threads to compute Fibonacci
    startButton.IsEnabled = false;
    outputTextBox.Text = "Starting Task to calculate Fibonacci(40)\r\n";

    // create Task to perform Fibonacci(46) calculation in a thread
    Task<long> e1 = Task.Run(() => StartFibonacciAsync(40));

    outputTextBox.AppendText("Starting Task to calculate Fibonacci(30)\r\n");

    // create Task to perform Fibonacci(45) calculation in a thread
    Task<long> e2 = Task.Run(() => StartFibonacciAsync(30));
    // await all the two Tasks to complete
    await Task.WhenAll<long>(e1, e2);

    //display total time for calculations
    outputTextBox.AppendText(String.Format(
        "Total calculation time = {0} ms\r\n",
         Math.Max(e1.Result, e2.Result)));
    startButton.IsEnabled = true;
}
```

Waits both Tasks to compute Fibonacci. The fastest running Task returns first its output to the GUI

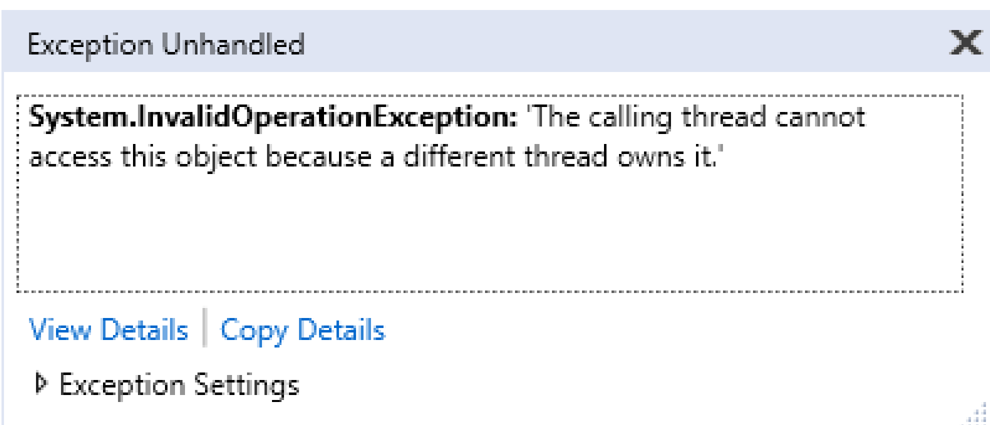
StartButton_ClickV2 runs on the **UI thread and it allows direct access to WPF controls and their properties**

15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

We must use **`Dispatcher.BeginInvoke()`** or **`Dispatcher.Invoke()`** in method **StartFibonacciAsync**() because it **runs on a thread different than the UI-thread in case this method is called from `StartButton_ClickV2()`**. For faster response of the GUI we **`await`** method **`Dispatcher.BeginInvoke.`**

In case of calling this method from **`StartButton_ClickV1()`** there is **no need to use** **`Dispatcher.BeginInvoke()`** because it runs on the UI thread.

In case we attempt to access directly objects and their properties in the GUI from a thread different from the UI thread the application throws an **`InvalidOperationException`**.



15.12.2 Asynchronous Execution of Two Compute– Intensive Tasks

```
async Task<long> StartFibonacciAsync(int n)
{  //runs on a TPL thread, not the UI thread
    Stopwatch sw = new Stopwatch();
    sw.Start();
    // execute task on the UI thread
    await outputTextBox.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
                        new Action<int>((int e) =>
                            outputTextBox.Text += String.Format("Calculating Fibonacci({0})\r\n", e)),
                        n);

    long fibonacciValue = await Task.FromResult<long>( Fibonacci(n)); //not using a TPL thread
    // time completion calculation
    long finishTime = sw.ElapsedMilliseconds;
    // execute task on the UI thread
    await outputTextBox.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
                        new Action<int, long>((int e, long l) =>
                            outputTextBox.Text += String.Format("Fibonacci({0}) = {1}", e, l)),
                        n, fibonacciValue);
    await outputTextBox.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
                        new Action<long>((e) =>
                            outputTextBox.Text += String.Format("   Calculation time = {0} ms\r\n", e)),
                        finishTime);
    sw.Reset();
    return finishTime;
} // end method StartFibonacci
```

StartFibonacciAsync runs with **Task.Run()** on a thread **different** than the **UI thread**

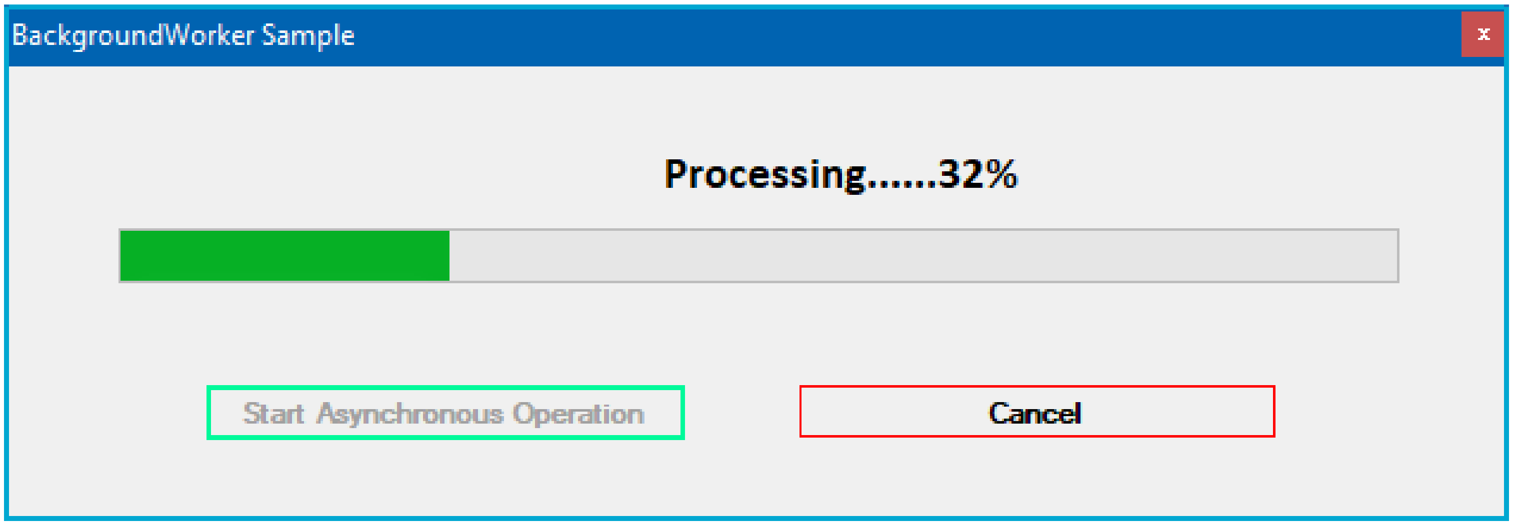
Run tasks on the UI thread

Compute Fibonacci on the same thread

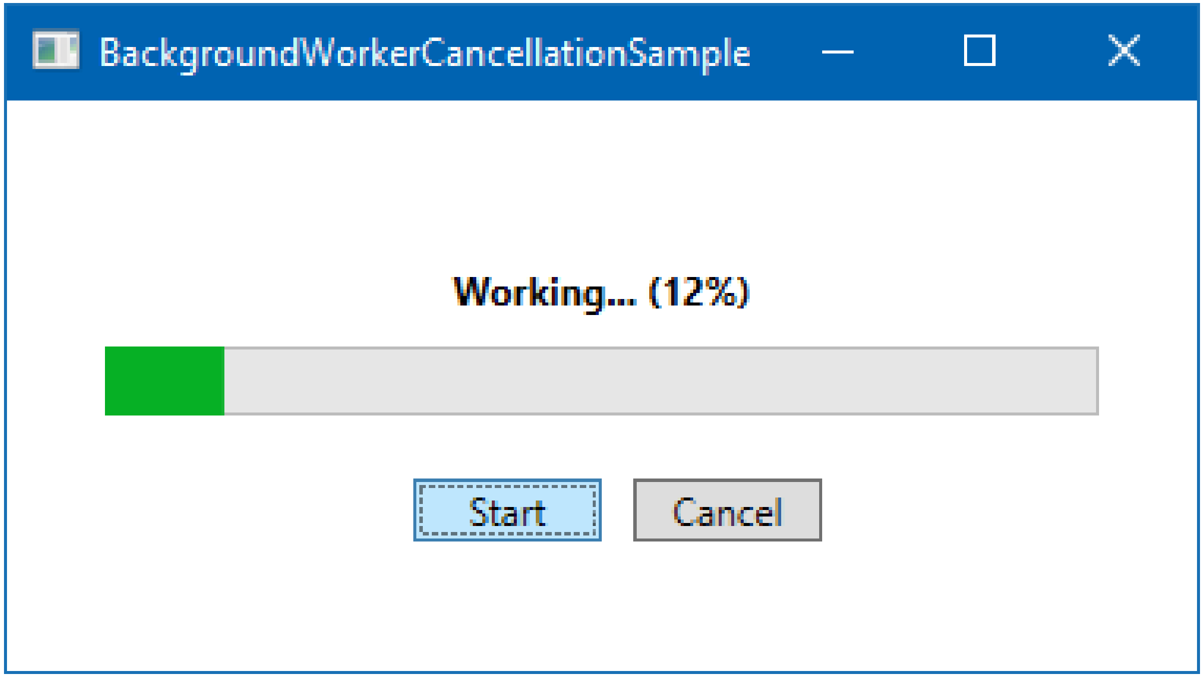
15.13 BackgroundWorker

A basic Windows application runs on a single thread usually referred to as UI thread. This UI thread is responsible for creating/painting all the controls and upon which the code execution takes place. So when you are running a long-running task (i.e., data intensive database operation or processing some 100s of bitmap images), the UI thread locks up and the UI application turns white (remember the UI thread was responsible to paint all the controls) rendering your application to a not Responding state.

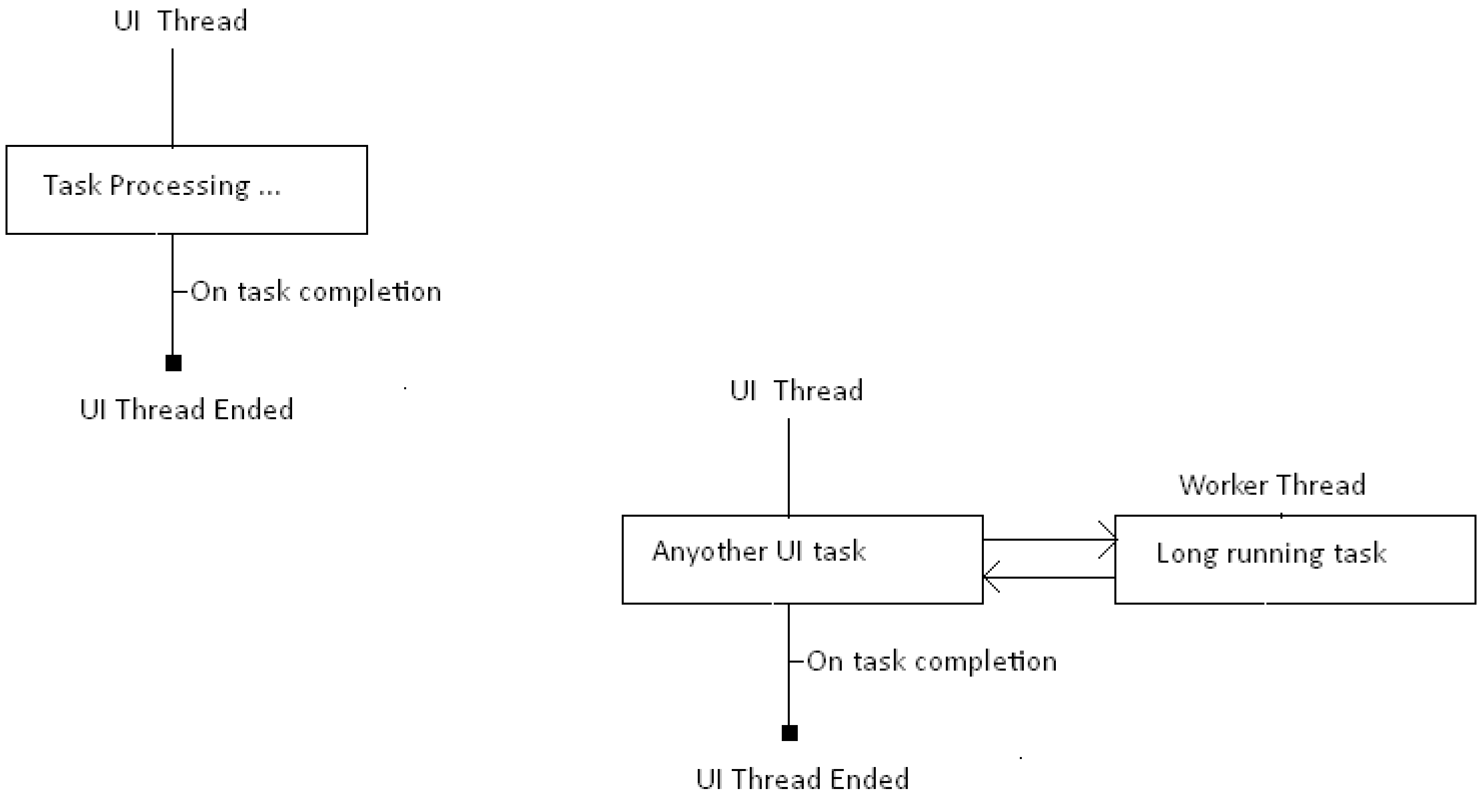
15.13.1 BackgroundWorker in Windows Forms



15.13 BackgroundWorker in WPF



15.13 BackgroundWorker



15.13 BackgroundWorker

The steps to use `BackgroundWorker` are:

1. **Create a `BackgroundWorker` object.**
2. **Tell the `BackgroundWorker` object what task to run on the background thread (the `DoWork` method).**
3. **Tell it what function to run on the UI thread when the work is complete (the `RunWorkerCompleted` method).**
4. **Start a process on the worker thread `bgWorker.RunWorkerAsync();`**

15.13 BackgroundWorker

```
//BackgroundWorker component is available from the Toolbox
BackgroundWorker bgWorker = new BackgroundWorker();
// Hook up the appropriate events.
// The events may be hooked from the Properties window
bgWorker.DoWork +=
                  new DoWorkEventHandler(bgWorker_DoWork);
bgWorker.RunWorkerCompleted +=
                  new RunWorkerCompletedEventHandler
                            (bgWorker_RunWorkerCompleted);
```

15.13 BackgroundWorker

```
void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{// Never access UI objects from the DoWork method
  e.Result = "Task Completed...“; // some object
}
void bgWorker_RunWorkerCompleted(object sender,
                                RunWorkerCompletedEventArgs e)
{ if (e.Cancelled)
    {lblStatus.Text = "Task Cancelled.„;}
    else if (e.Error != null)
     {lblStatus.Text = "Error … ";}
         else // Everything completed normally.
            { lblStatus.Text = e.Result as String;}
}
```

15.13 BackgroundWorker

The `BackgroundWorker` uses the thread-pool, which recycles threads to avoid recreating them for each new task. This means one should never call `Abort` on a `BackgroundWorker` thread.

Note:

Never access UI objects on a thread that didn't create them. It means you cannot use a code such as this

```
lblProgressIndicator.Text = "Processing file...20%";
```

in a Thread different from the UI thread or from the `DoWork` method of the `BackgroundWorker`

15.13 BackgroundWorker

The `BackgroundWorker` object resolves this problem by giving us a `ReportProgress` method which can be called from the background thread's `DoWork` method. This will cause the `ProgressChanged` event to fire on the UI thread. Now we can access the UI objects on their thread and do what we want (*In case of attached Sample code for the* `BackgroundWorkerSimple` *project, setting the label text status*).

15.13 BackgroundWorker

`BackgroundWorker` **also provides a** `RunWorkerCompleted` **event which fires after the** `DoWork` **event handler has done its job. Handling** `RunWorkerCompleted` **is not mandatory, but one usually does so in order to query any exception that was thrown in** `DoWork`**.**

15.13 BackgroundWorker

Furthermore, code within a `RunWorkerCompleted` event handler is able to update Windows Forms and WPF controls without explicit marshalling, while code within the `DoWork` event handler `cannot do so`.

15.13 BackgroundWorker

To add support for `progress reporting`:

1. Set the `WorkerReportsProgress` property to `true`.
2. Periodically call `ReportProgress` from within the `DoWork` event handler with a "`percentage complete`" value. For instance, as

```
bgWorker.ReportProgress(i);
```

3. Handle the `ProgressChanged` event, querying its event argument's `ProgressPercentage` property as shown on the following slide

15.13 BackgroundWorker

```
bgWorker.ProgressChanged += new
 ProgressChangedEventHandler
                    (bgWorker_ProgressChanged);
bgWorker.WorkerReportsProgress = true;
```

15.13 BackgroundWorker

```
void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{
// The sender is the BackgroundWorker object we need it to
// report progress and check for cancellation.
//NOTE : Never play with the UI thread here...
    for (int i = 0; i < 100; i++)
    {
     // Periodically report progress to the main thread so that it can
     // update the UI.  In most cases you'll just need to send an
     // integer that will update a ProgressBar
         bgWorker.ReportProgress(i);
     }
     //Report 100% completion on operation completed
     bgWorker.ReportProgress(100);
 }
```

15.13 BackgroundWorker

```
public void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    // This function fires on the UI thread so it's safe to edit
    // the UI control directly, no need to use Control.Invoke
    // Update the progressBar with the integer supplied to us from the
    // ReportProgress() function.
    progressBar.Value = e.ProgressPercentage;
    lblProgressIndicator.Text = "Processing..." + progressBar.Value.ToString() + "%";
}
```

Code in the **`ProgressChanged`** event handler is free to interact with UI controls just as with **`RunWorkerCompleted.`** This is typically where you will update a progress bar .

15.13 BackgroundWorker

To add support for cancellation:

1. **Set the `WorkerSupportsCancellation` property to `true`.**

```
bgWorker.WorkerSupportsCancellation = true;
```

2. **Periodically check the `CancellationPending` property from within the `DoWork` event handler- if `true`, set the event argument's `Cancel` property to `true`, and `return`. (The worker can set `Cancel` = `true` and exit without prompting via `CancellationPending`, if it decides the job's too difficult and it can't go on).**
3. **Call `CancelAsync` to request cancellation. This code is handled on click of the `Cancel` button.**

15.13 BackgroundWorker

```
void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{    for (int i = 0; i < 100; i++)
      {
        Thread.Sleep(100);
        bgWorker.ReportProgress(i);
         // Periodically check if a cancellation request is pending.
         // If the user executes the command bgWorker.CancelAsync()
         // it sets the CancellationPending to true.
         // You must check this flag in here and react to it.
         // We react to it by setting e.Cancel to true and leavingm
         if (bgWorker.CancellationPending)
          {
            // Set the e.Cancel flag so that the WorkerCompleted event
            // knows that the process was cancelled.
            e.Cancel = true;
            bgWorker.ReportProgress(0);
            return;
           }
        } //Report 100% completion on operation completed
        bgWorker.ReportProgress(100);
}
```

15.13 BackgroundWorker

```
 if (bgWorker.CancellationPending)
  {
     // Set the e.Cancel flag so that the WorkerCompleted event
     // knows that the process was cancelled.
     e.Cancel = true;
     bgWorker.ReportProgress(0);
     return;
  }
// bgWorker.CancellationPending is set by a call
 bgWorker.CancelAsync();
```

15.13 BackgroundWorker

Essential `BackgroundWorker` properties of `DoWorkEventArgs e`

Usage: Contains `e.Argument` and `e.Result`

- **`e.Argument` Usage: Get the parameter `param` reference received by `RunWorkerAsync`.**
- **`e.Result` Usage: Set the `BackgroundWorker` processing. object. Will be available to the RunWorkerCompleted eventhandler.**

```
bgWorker.RunWorkerAsync(); // or
bgWorker.RunWorkerAsync(object param);
```

Usage: Called to start a process on the worker thread,

`param` is passed to the background operation to be executed in the DoWork event handler

15.13 BackgroundWorker summary

The basics:

1a. Create an instance of the class (new BackgroundWorker())

2a. Hook up to its events (DoWork, RunWorkerCompleted)

3a. Tell it you want to do some background work (RunWorkerAsync)

4a. It raises the event. In your DoWorkEventHandler method do your background stuff (without touching GUI of course)

5a. When done, your RunWorkerCompletedEventHandler method will run on the GUI

15.13 BackgroundWorker summary

Add a value [Pass state in, cancel from job]:

3b. Optionally pass an object to the worker method (RunWorkerAsync, DoWorkEventArgs.Argument)

4b. Optionally decide in your background work to cancel (DoWorkEventArgs.Cancel=true)

5b. In your `RunWorkerCompletedEventHandler` method, check whether the operation was cancelled (RunWorkerCompletedEventArgs.Cancelled)

15.13 BackgroundWorker summary

Get a result back:

4c. From your background work, optionally pass a result to the GUI when it's finished (DoWorkEventArgs.Result)

5c. Check for it (RunWorkerCompletedEventArgs.Result)

15.13 BackgroundWorker summary

Progress notification

2b. Additionally hook up to a progress event (ProgressChanged) if you configure the object to allow it (WorkerReportsProgress)

4d. From your worker method, let the object know the percentage (ReportProgress) optionally passing some state (ReportProgress)

6. In your ProgressChangedEventHandler method that runs on the GUI, check the percentage (ProgressChangedEventArgs.ProgressPercentage) and optionally the state given (ProgressChangedEventArgs.UserState)

15.13 BackgroundWorker summary

Finally [Cancel from outside the job]

2c. Optionally Configure the object to allow cancellation (WorkerSupportsCancellation) other than just from the background worker method itself

4e. In your worker method check periodically if a cancel has been issued (CancellationPending) and if it has proceed as 4b

7. From any place in the code, ask the worker to cancel (CancelAsync)