



Timers Prior to the Async/Await Pattern of C# 5.0

CHAPTER 19 INTRODUCED THE USE of `Task.Delay()` when a timer was required. For scenarios prior to .NET 4.5, several timer classes are available, including `System.Windows.Forms.Timer`, `System.Timers.Timer`, and `System.Threading.Timer`.

The development team designed `System.Windows.Forms.Timer` specifically for use within a rich client user interface. Programmers can drag it onto a form as a nonvisual control and regulate the behavior from within the Properties window. Most importantly, it will always safely fire an event from a thread that can interact with the user interface.

The other two timers are very similar. `System.Timers.Timer` is a wrapper for `System.Threading.Timer`, abstracting and layering on functionality. Specifically, `System.Threading.Timer` does not derive from `System.ComponentModel.Component`, and therefore, you cannot use it as a component within a component container, something that implements `System.ComponentModel.IContainer`. Another difference is that `System.Threading.Timer` enables the passing of state, an object parameter, from the call to start the timer and then into the call that fires the timer notification. The remaining differences simply concern API usability, with `System.Timers.Timer` supporting a synchronization object and having calls that are slightly more intuitive. Both `System.Timers.Timer` and `System.Threading.Timer` are designed for use in server-type processes,

but `System.Timers.Timer` includes a synchronization object to allow it to interact with the UI. Furthermore, both timers use the system thread pool. Table D.1 provides an overall comparison of the various timers.

TABLE D.1: Overview of the Various Timer Characteristics

Feature Description	<code>System.Timers.Timer</code>	<code>System.Threading.Timer</code>	<code>System.Windows.Forms.Timer</code>
Supports adding and removing listeners after the timer is instantiated	Yes	No	Yes
Supports callbacks on the user interface thread	Yes	No	Yes
Calls back from threads obtained from the thread pool	Yes	Yes	No
Supports drag-and-drop in the Windows Forms Designer	Yes	No	Yes
Suitable for running in a multithreaded server environment	Yes	Yes	No
Includes support for passing arbitrary state from the timer initialization to the callback	No	Yes	No
Implements <code>IDisposable</code>	Yes	Yes	Yes
Supports on-off callbacks as well as periodic repeating callbacks	Yes	Yes	Yes
Accessible across application domain boundaries	Yes	Yes	Yes
Supports <code>IComponent</code> ; hostable in an <code>IContainer</code>	Yes	No	Yes

Using `System.Windows.Forms.Timer` is a relatively obvious choice for user interface programming with Windows Forms. The only caution is that a long-running operation on the user interface thread may delay the arrival

of a timer's expiration.¹ Choosing between the other two options is less obvious, and generally, the difference between the two is insignificant. If hosting within an `IContainer` is necessary, `System.Timers.Timer` is the right choice. However, if no specific `System.Timers.Timer` feature is required, choose `System.Threading.Timer` by default, simply because it is a slightly lighter-weight implementation.

Listing D.1 and Listing D.2 provide sample code for using `System.Timers.Timer` and `System.Threading.Timer`, respectively. Their code is very similar, including the fact that both support instantiation within a `using` statement because both support `IDisposable`. The output for both listings is identical, and it appears in Output D.1. The purpose of each is to display a timestamp in association with a counting value indicating the number of times the timer fired. Once complete, the output verifies that the timer thread is not the same as the Main thread along with the final value of the count.

LISTING D.1: Using `System.Timers.Timer`

```
using System;
using System.Timers;
using System.Threading;
// Because Timer exists in both the System.Timers and
// System.Threading namespaces, you disambiguate "Timer"
// using an alias directive.
using Timer = System.Timers.Timer;

class UsingSystemTimersTimer
{
    private static int _Count=0;
    private static readonly ManualResetEvent _ResetEvent =
        new ManualResetEvent(false);
    private static int _AlarmThreadId;

    public static void Main()
    {
        using( Timer timer = new Timer() )
        {
            // Initialize Timer
            timer.AutoReset = true;
            timer.Interval = 1000;
            timer.Elapsed +=
                new ElapsedEventHandler(Alarm);

            timer.Start();
```

1. In theory, a similar delay is possible with timers that depend on a thread pool as well because the thread pool may already be busy.

4 ■ Appendix D: Timers

```
        // Wait for Alarm to fire for the 10th time.
        _ResetEvent.WaitOne();
    }

    // Verify that the thread executing the alarm
    // Is different from the thread executing Main
    if(_AlarmThreadId ==
        Thread.CurrentThread.ManagedThreadId)
    {
        throw new ApplicationException(
            "Thread Ids are the same.");
    }
    if(_Count < 9)
    {
        throw new ApplicationException(
            " _Count < 9");
    };

    Console.WriteLine(
        "(Alarm Thread Id) {0} != {1} (Main Thread Id)",
        _AlarmThreadId,
        Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine(
        "Final Count = {0}", _Count);
}

static void Alarm(
    object sender, ElapsedEventArgs eventArgs)
{
    _Count++;

    Console.WriteLine("{0}:- {1}",
        eventArgs.SignalTime.ToString("T"),
        _Count);

    if (_Count >= 9)
    {
        _AlarmThreadId =
            Thread.CurrentThread.ManagedThreadId;
        _ResetEvent.Set();
    }
}
}
```

In Listing D.1, you have using directives for both `System.Threading` and `System.Timers`. This makes the `Timer` type ambiguous. Therefore, use an alias to explicitly associate `Timer` with `System.Timers.Timer`.

One noteworthy characteristic of `System.Threading.Timer` is that it takes the callback delegate and interval within the constructor.

LISTING D.2: Using `System.Threading.Timer`

```

using System;
using System.Threading;

class UsingSystemThreadingTimer
{
    private static int _Count=0;
    private static readonly AutoResetEvent _ResetEvent =
        new AutoResetEvent(false);
    private static int _AlarmThreadId;

    public static void Main()
    {
        // Timer(callback, state, dueTime, period)
        using( Timer timer =
            new Timer(Alarm, null, 0, 1000) )
        {
            // Wait for Alarm to fire for the 10th time.
            _ResetEvent.WaitOne();
        }

        // Verify that the thread executing the alarm
        // Is different from the thread executing Main
        if(_AlarmThreadId ==
            Thread.CurrentThread.ManagedThreadId)
        {
            throw new ApplicationException(
                "Thread Ids are the same.");
        }
        if(_Count < 9)
        {
            throw new ApplicationException(
                " _Count < 9");
        };

        Console.WriteLine(
            "(Alarm Thread Id) {0} != {1} (Main Thread Id)",
            _AlarmThreadId,
            Thread.CurrentThread.ManagedThreadId);
        Console.WriteLine(
            "Final Count = {0}", _Count);
    }

    static void Alarm(object state)
    {
        _Count++;
    }
}

```

6 ■ Appendix D: Timers

```
Console.WriteLine("{0}:- {1}",
    DateTime.Now.ToString("T"),
    _Count);

if (_Count >= 9)
{
    _AlarmThreadId =
        Thread.CurrentThread.ManagedThreadId;
    _ResetEvent.Set();
}
}
```

OUTPUT D.1

```
12:19:36 AM:- 1
12:19:37 AM:- 2
12:19:38 AM:- 3
12:19:39 AM:- 4
12:19:40 AM:- 5
12:19:41 AM:- 6
12:19:42 AM:- 7
12:19:43 AM:- 8
12:19:44 AM:- 9
(Alarm Thread Id) 4 != 1 (Main Thread Id)
Final Count = 9
```

You can change the interval or time due after instantiation on `System.Threading.Timer` via the `Change()` method. However, you cannot change the callback listeners after instantiation. Instead, you must create a new instance.