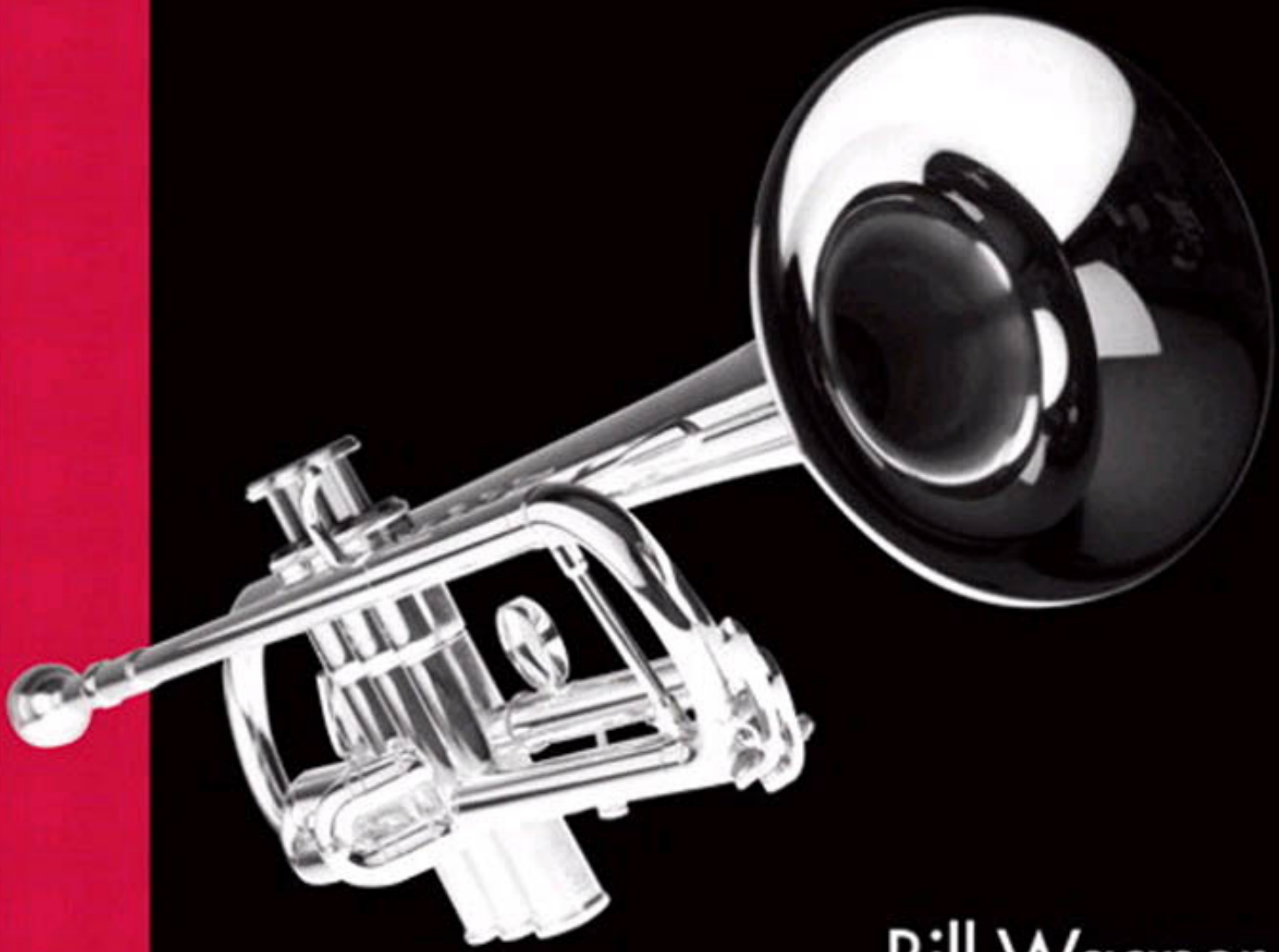
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

MORE *Effective* C#

50 Specific Ways to Improve Your C#



Bill Wagner

Praise for *More Effective C#*

"Shining a bright light into many of the dark corners of C# 3.0, this book not only covers the 'how,' but also the 'why,' arming the reader with many field-tested methods for wringing the most from the new language features, such as LINQ, generics, and multithreading. If you are serious about developing with the C# language, you need this book."

—Bill Craun, Principal Consultant, Ambassador Solutions, Inc.

"*More Effective C#* is an opportunity to work beside Bill Wagner. Bill leverages his knowledge of C# and distills his expertise down to some very real advice about programming and designing applications that every serious Visual C# user should know. *More Effective C#* is one of those rare books that doesn't just regurgitate syntax, but teaches you *how* to use the C# language."

—Peter Ritchie, Microsoft MVP: Visual C#

"*More Effective C#* is a great follow-up to Bill Wagner's previous book. The extensive C# 3.0 and LINQ coverage is extremely timely!"

—Tomas Restrepo, Microsoft MVP: Visual C++, .NET, and Biztalk Server

"As one of the current designers of C#, it is rare that I learn something new about the language by reading a book. *More Effective C#* is a notable exception. Gently blending concrete code and deep insights, Bill Wagner frequently makes me look at C# in a fresh light—one that really makes it shine. *More Effective C#* is at the surface a collection of very useful guidelines. Look again. As you read through it, you'll find that you acquire more than just the individual pieces of advice; gradually you'll pick up on an approach to programming in C# that is thoughtful, beautiful, and deeply pleasant. While you can make your way willy-nilly through the individual guidelines, I do recommend reading the whole book—or at least not skipping over the chapter introductions before you dive into specific nuggets of advice. There's perspective and insight to be found there that in itself can be an important guide and inspiration for your future adventures in C#."

—Mads Torgersen, Program Manager, Visual C#, Microsoft

"Bill Wagner has written an excellent book outlining the best practices for developers who work with the C# language. By authoring *More Effective C#*, he has again established himself as one of the most important voices in the C# community. Many of us already know how to use C#. What we need is advice on how to hone our skills so that we can become wiser programmers. There is no more sophisticated source of information on how to become a first-class C# developer than Bill Wagner's book. Bill is intelligent, thoughtful, experienced, and skillful. By applying the lessons from this book to your own code, you will find many ways to polish and improve the work that you produce."

—Charlie Calvert, Community Program Manager, Visual C#, Microsoft

Simply return from the background thread procedure, and handle the error in the event handler for the foreground results.

Earlier I said that I often use `BackgroundWorker` in classes that aren't the `Form` class, and even in non-Windows Forms applications, such as services or Web services. This works fine, but it does have some caveats. When `BackgroundWorker` determines that it is running in a Windows Forms application and the form is visible, the `ProgressChanged` and `RunWorkerCompleted` events are marshaled to the graphical user interface (GUI) thread via a marshaling control and `Control.BeginInvoke` (see Item 16 later in this chapter). In other scenarios, those delegates are simply called on a free thread pool thread. As you will see in Item 16, that behavior may affect the order in which events are received.

Finally, because `BackgroundWorker` is built on `QueueUserWorkItem`, you can reuse `BackgroundWorker` for multiple background requests. You need to check the `IsBusy` property of `BackgroundWorker` to see whether `BackgroundWorker` is currently running a task. When you need to have multiple background tasks running, you can create multiple `BackgroundWorker` objects. Each will share the same thread pool, so you have multiple tasks running just as you would with `QueueUserWorkItem`. You need to make sure that your event handlers use the correct sender property. This practice ensures that the background threads and foreground threads are communicating correctly.

`BackgroundWorker` supports many of the common patterns that you will use when you create background tasks. By using it you can reuse that implementation in your code, adding any of those patterns as needed. You don't have to design your own communication protocols between foreground and background threads.

Item 13: Use lock() as Your First Choice for Synchronization

Threads need to communicate with each other. Somehow, you need to provide a safe way for various threads in your application to send and receive data. However, sharing data between threads introduces the potential for data integrity errors in the form of synchronization issues. Therefore, you need to be certain that the current state of every shared data item is consistent. You achieve this safety by using **synchronization primitives** to protect access to the shared data. Synchronization primitives ensure that the current thread is not interrupted until a critical set of operations is completed.

There are many primitives available in the .NET BCL that you can use to safely ensure that access to shared data is synchronized. Only one pair of them—`Monitor.Enter()` and `Monitor.Exit()`—was given special status in the C# language. `Monitor.Enter()` and `Monitor.Exit()` implement a **critical section** block. Critical sections are such a common synchronization technique that the language designers added support for them using the `lock()` statement. You should follow that example and make `lock()` your primary tool for synchronization.

The reason is simple: The compiler generates consistent code, but you may make mistakes some of the time. The C# language introduces the *lock* keyword to control synchronization for multithreaded programs. The lock statement generates exactly the same code as if you used `Monitor.Enter()` and `Monitor.Exit()` correctly. Furthermore, it's easier and it automatically generates all the exception-safe code you need.

However, under two conditions `Monitor` gives you necessary control that you can't get when you use `lock()`. First, be aware that `lock` is lexically scoped. This means that you can't enter a `Monitor` in one lexical scope and exit it in another when using the `lock` statement. Thus, you can't enter a `Monitor` in a method and exit it inside a lambda expression defined in that method (see Item 41, Chapter 5). The second reason is that `Monitor.Enter` supports a time-out, which I cover later in this item.

You can lock any reference type by using the `lock` statement:

```
public int TotalValue
{
    get
    {
        lock(syncHandle)
        {
            return total;
        }
    }
}

public void IncrementTotal()
{
    lock (syncHandle)
    {
        total++;
    }
}
```

The `lock` statement gets the exclusive monitor for an object and ensures that no other thread can access the object until the lock is released. The preceding sample code, using `lock()`, generates the same IL as the following version, using `Monitor.Enter()` and `Monitor.Exit()`:

```
public void IncrementTotal()
{
    object tmpObject = syncHandle;
    System.Threading.Monitor.Enter(tmpObject);
    try
    {
        total++;
    }
    finally
    {
        System.Threading.Monitor.Exit(tmpObject);
    }
}
```

The `lock` statement provides many checks that help you avoid common mistakes. It checks that the type being locked is a reference type, as opposed to a value type. The `Monitor.Enter` method does not include such safeguards. This routine, using `lock()`, doesn't compile:

```
public void IncrementTotal()
{
    lock (total) // compiler error: can't lock value type
    {
        total++;
    }
}
```

But this does:

```
public void IncrementTotal()
{
    // really doesn't lock total.
    // locks a box containing total.
    Monitor.Enter(total);
    try
    {
        total++;
    }
```

```
    finally
    {
        // Might throw exception
        // unlocks a different box containing total
        Monitor.Exit(total);
    }
}
```

`Monitor.Enter()` compiles because its official signature takes a `System.Object`. You can coerce `total` into an object by boxing it. `Monitor.Enter()` actually locks the box containing `total`. That's where the first bug lurks. Imagine that thread 1 enters `IncrementTotal()` and acquires a lock. Then, while incrementing `total`, the second thread calls `IncrementTotal()`. Thread 2 now enters `IncrementTotal()` and acquires the lock. It succeeds in acquiring a different lock, because `total` gets put into a different box. Thread 1 has a lock on one box containing the value of `total`. Thread 2 has a lock on another box containing the value of `total`. You've got extra code in place, and no synchronization.

Then you get bitten by the second bug: When either thread tries to release the lock on `total`, the `Monitor.Exit()` method throws a `SynchronizationLockException`. That's because `total` goes into yet another box to coerce it into the method signature for `Monitor.Exit`, which also expects a `System.Object` type. When you release the lock on this box, you unlock a resource that is different from the resource that was used for the lock. `Monitor.Exit()` fails and throws an exception.

Of course, some bright soul might try this:

```
public void IncrementTotal()
{
    // doesn't work either:
    object lockHandle = total;
    Monitor.Enter(lockHandle);
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(lockHandle);
    }
}
```

This version doesn't throw any exceptions, but neither does it provide any synchronization protection. Each call to `IncrementTotal()` creates a new box and acquires a lock on that object. Every thread succeeds in immediately acquiring the lock, but it's not a lock on a shared resource. Every thread wins, and `total` is not consistent.

There are subtler errors that `lock` also prevents. `Enter()` and `Exit()` are two separate calls, so you can easily make the mistake of acquiring and releasing different objects. This action may cause a `SynchronizationLockException`. But if you happen to have a type that locks more than one synchronization object, it's possible to acquire two different locks in a thread and release the wrong one at the end of a critical section.

The `lock` statement automatically generates exception-safe code, something many of us humans forget to do. Also, it generates more-efficient code than `Monitor.Enter()` and `Monitor.Exit()`, because it needs to evaluate the target object only once. So, by default, you should use the `lock` statement to handle the synchronization needs in your C# programs.

However, there is one limitation to the fact that `lock` generates the same MSIL as `Monitor.Enter( )`. The problem is that `Monitor.Enter()` waits forever to acquire the lock. You have introduced a possible deadlock condition. In large enterprise systems, you may need to be more defensive in how you attempt to access critical resources. `Monitor.TryEnter( )` lets you specify a time-out for an operation and attempt a workaround when you can't access a critical resource.

```
public void IncrementTotal()
{
    if (!Monitor.TryEnter(syncHandle, 1000)) // wait 1 second
        throw new PreciousResourceException
            ("Could not enter critical section");
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(syncHandle);
    }
}
```

You can wrap this technique in a handy little generic class:

```
public sealed class LockHolder<T> : IDisposable
    where T : class
{
    private T handle;
    private bool holdsLock;

    public LockHolder(T handle, int milliSecondTimeout)
    {
        this.handle = handle;
        holdsLock = System.Threading.Monitor.TryEnter(
            handle, milliSecondTimeout);
    }

    public bool LockSuccessful
    {
        get { return holdsLock; }
    }

    #region IDisposable Members
    public void Dispose()
    {
        if (holdsLock)
            System.Threading.Monitor.Exit(handle);
        // Don't unlock twice
        holdsLock = false;
    }
    #endregion
}
```

You would use this class in the following manner:

```
object lockHandle = new object();

using (LockHolder<object> lockObj = new LockHolder<object>
    (lockHandle, 1000))
{
    if (lockObj.LockSuccessful)
    {
        // work elided
    }
}
// Dispose called here.
```