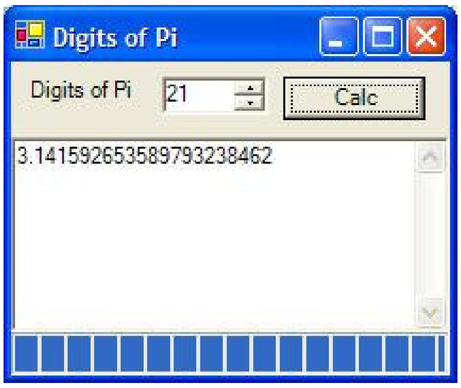
Safe, Simple Multithreading in Windows Forms, Part 1

Chris Sells

June 28, 2002

Download the AsynchCalcPi.exe sample.

It all started innocently enough. I found myself needing to calculate the area of a circle for the first time in .NET. This called, of course, for an accurate representation of pi. System.Math.PI is handy, but since it only provides 20 digits of precision, I was worried about the accuracy of my calculation (I really needed 21 digits to be absolutely comfortable). So, like any programmer worth their salt, I forgot about the problem I was actually trying to solve and I wrote myself a program to calculate pi to any number of digits that I felt like. What I came up with is shown in Figure 1.

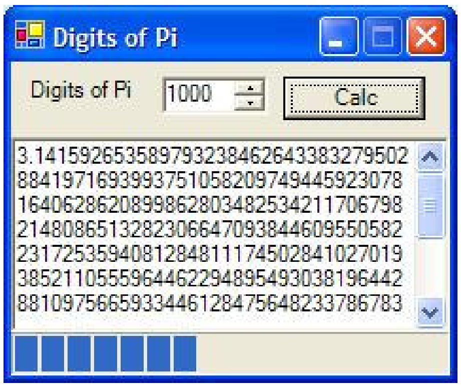


```
3.141592653589793238462
```

Figure 1. Digits of Pi application

Progress on Long-Running Operations

While most applications don't need to calculate digits of pi, many kinds of applications need to perform long-running operations, whether it's printing, making a Web service call, or calculating interest earnings on a certain billionaire in the Pacific Northwest. Users are generally content to wait for such things, often moving to something else in the meantime, so long as they can see that progress is being made. That's why even my little application has a progress bar. The algorithm I'm using calculates pi nine digits at a time. As each new set of digits are available, my program keeps the text updated and moves the progress bar to show how we're coming along. For example, Figure 2 shows progress on the way to calculating 1000 digits of pi (if 21 digits are good, than 1000 must be better).



```
3.141592653589793238462643383279502
8841971693993751058209749445923078
1640628620899862803482534211706798
2148086513282306647093844609550582
2317253594081284811174502841027019
3852110555964462294895493038196442
8810975665933446128475648233786783
```

Figure 2. Calculating pi to 1000 digits

The following shows how the user interface (UI) is updated as the digits of pi are calculated:

```
void ShowProgress(string pi, int totalDigits, int digitsSoFar) {
  _pi.Text = pi;
  _piProgress.Maximum = totalDigits;
  _piProgress.Value = digitsSoFar;
}

void CalcPi(int digits) {
  StringBuilder pi = new StringBuilder("3", digits + 2);

  // Show progress
  ShowProgress(pi.ToString(), digits, 0);

  if( digits > 0 ) {
    pi.Append(".");

    for( int i = 0; i < digits; i += 9 ) {
      int nineDigits = NineDigitsOfPi.StartingAt(i+1);
      int digitCount = Math.Min(digits - i, 9);
      string ds = string.Format("{0:D9}", nineDigits);
      pi.Append(ds.Substring(0, digitCount));

      // Show progress
      ShowProgress(pi.ToString(), digits, i + digitCount);
    }
  }
}
```

Everything was going along fine until, in the middle of actually calculating pi to 1000 digits, I switched away to do something else and then switched back. What I saw is shown in Figure 3.

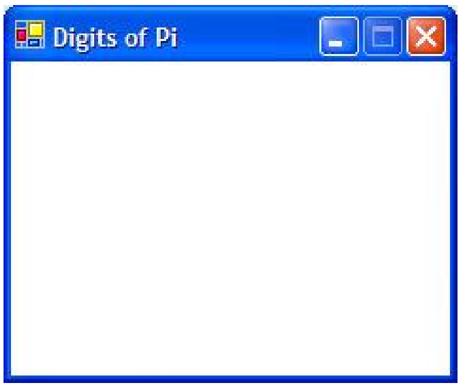


Figure 3. No paint event for you!

The problem, of course, is that my application is single-threaded, so while the thread is calculating pi, it can't also be drawing the UI. I didn't run into this before because when I set the **TextBox.Text** and **ProgressBar.Value** properties, those controls would force their painting to happen immediately as part of setting the property (although I noticed that the progress bar was better at this than the text box). However, once I put the application into the background and then the foreground again, I need to paint the entire client area, and that's a Paint event for the form. Since no other event is going to be processed until we return from the event we're already processing (that is, the **Click** event on the Calc button), we're out of luck in terms of seeing any further progress. What I really needed to do was free the UI thread for doing UI work and handle the long-running process in the background. For this, I need another thread.

Asynchronous Operations

My current synchronous **Click** handler looked like this:

```
void _calcButton_Click(object sender, EventArgs e) {
  CalcPi((int)_digits.Value);
}
```

Recall that the issue is until **CalcPi** returns, the thread can't return from our **Click** handler, which means the form can't handle the **Paint** event (or any other event, for that matter). One way to handle this is to start another thread, like so:

```
using System.Threading;
Ã‚Â…
int _digitsToCalc = 0;

void CalcPiThreadStart() {
  CalcPi(_digitsToCalc);
}

void _calcButton_Click(object sender, EventArgs e) {
  _digitsToCalc = (int)_digits.Value;
  Thread piThread = new Thread(new ThreadStart(CalcPiThreadStart));
  piThread.Start();
}
```

Now, instead of waiting for **CalcPi** to finish before returning from the button **Click** event, I'm creating a new thread and asking it to start. The **Thread.Start** method will schedule my new thread as ready to start and then return immediately, allowing our UI thread to get back to its own work. Now, if the user wants to interact with the application (put it in the background, move it to the foreground, resize it, or even close it), the UI thread is free to handle all of those events while the worker thread calculates pi at its own pace. Figure 4 shows the two threads doing the work.

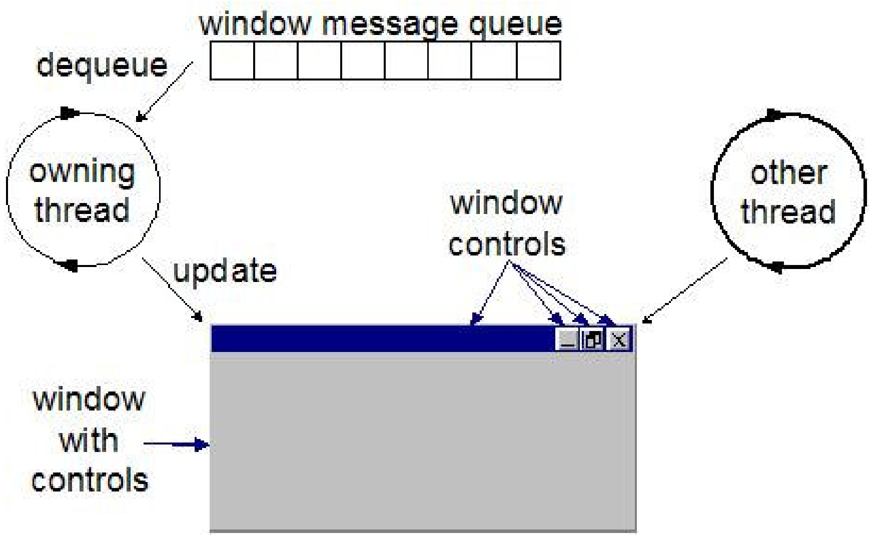


Figure 4. Naive multithreading

You may have noticed that I'm not passing any arguments to the worker thread's entry point—**CalcPiThreadStart**. Instead, I'm tucking the number of digits to calculate into a field, `_digitsToCalc`, calling the thread entry point, which is calling **CalcPi** in turn. This is kind of a pain, which is one of the reasons that I prefer delegates for asynchronous work. Delegates support taking arguments, which saves me the hassle of an extra temporary field and an extra function between the functions I want to call.

If you're not familiar with delegates, they're really just objects that call static or instance functions. In C#, they're declared using function declaration syntax. For example, a delegate to call **CalcPi** looks like this:

```
delegate void CalcPiDelegate(int digits);
```

Once I have a delegate, I can create an instance to call the **CalcPi** function synchronously like so:

```
void _calcButton_Click(object sender, EventArgs e) {
  CalcPiDelegate  calcPi = new CalcPiDelegate(CalcPi);
  calcPi((int)_digits.Value);
}
```

Of course, I don't want to call **CalcPi** synchronously; I want to call it asynchronously. Before I do that, however, we need to understand a bit more about how delegates work. My delegate declaration above declares a new class derived from **MultiCastDelegate** with three functions, **Invoke**, **BeginInvoke**, and **EndInvoke**, as shown here:

```
class CalcPiDelegate : MulticastDelegate {
  public void Invoke(int digits);
  public void BeginInvoke(int digits, AsyncCallback callback,
                          object asyncState);
  public void EndInvoke(IAsyncResult result);
}
```

When I created an instance of the **CalcPiDelegate** earlier and then called it like a function, I was actually calling the synchronous **Invoke** function, which in turn called my own **CalcPi** function. **BeginInvoke** and **EndInvoke**, however, are the pair of functions that allow you to invoke and harvest the results of a function call asynchronously. So, to have the **CalcPi** function called on another thread, I need to call **BeginInvoke** like so:

```
void _calcButton_Click(object sender, EventArgs e) {
  CalcPiDelegate  calcPi = new CalcPiDelegate(CalcPi);
  calcPi.BeginInvoke((int)_digits.Value, null, null);
}
```

Notice that we're passing nulls for the last two arguments of **BeginInvoke**. These are needed if we'd like to harvest the result from the function we're calling at some later date (which is also what **EndInvoke** is for). Since the **CalcPi** function updates the UI directly, we don't need anything but nulls for these two arguments. If you'd like the details of delegates, both synchronous and asynchronous, see .NET Delegates: A C# Bedtime Story.

At this point, I should be happy. I've got my application to combine a fully interactive UI that shows progress on a long-running operation. In fact, it wasn't until I realized what I was really doing that I became unhappy.

Multithreaded Safety

As it turned out, I had just gotten lucky (or unlucky, depending on how you characterize such things). Microsoft Windows® XP was providing me with a very robust implementation of the underlying windowing system on which Windows Forms is built. So robust, in fact, that it gracefully handled my violation of the prime directive of Windows programming—*Though shalt not operate on a window from other than its creating thread*. Unfortunately there's no guarantee that other, less robust implementations of Windows would be equally graceful given my bad manners.

The problem, of course, was of my own making. If you remember Figure 4, I had two threads accessing the same underlying window at the same time. However, because long-running operations are so common in Windows application, each UI class in Windows Forms (that is, every class that ultimately derives from System.Windows.Forms.Control) has a property that you can use from any thread so that you can access the window safely. The name of the property is **InvokeRequired**, which returns true if the calling thread needs to pass control over to the creating thread before calling a method on that object. A simple Assert in my **ShowProgress** function would have immediately shown me the error of my ways:

```
using System.Diagnostics;

void ShowProgress(string pi, int totalDigits, int digitsSoFar) {
  // Make sure we're on the right thread
  Debug.Assert(_pi.InvokeRequired == false);
  ...
}
```

In fact, the .NET documentation is quite clear on this point. It states, "There are four methods on a control that are safe to call from any thread: **Invoke**, **BeginInvoke**, **EndInvoke**, and **CreateGraphics**. For all other method calls, you should use one of the invoke methods to marshal the call to the control's thread." So, when I set the control properties, I'm clearly violating this rule. And from the names of the first three functions that I'm allowed to call safely (**Invoke**, **BeginInvoke**, and **EndInvoke**), it should be clear that I need to construct another delegate that will be executed in the UI thread. If I were worried about blocking my worker thread, like I was worried about blocking my UI thread, I'd need to use the asynchronous **BeginInvoke** and **EndInvoke**. However, since my worker thread exists only to service my UI thread, let's use the simpler, synchronous **Invoke** method, which is defined like this:

```
public object Invoke(Delegate method);
public object Invoke(Delegate method, object[] args);
```

The first overload of **Invoke** takes an instance of a delegate containing the method we'd like to call in the UI thread, but assumes no arguments. However, the function we want to call to update the UI, **ShowProgress**, takes three arguments, so we'll need the second overload. We'll also need another delegate for our **ShowProgress** method so that we can pass the arguments correctly. Here's how to use **Invoke** to make sure that our calls to **ShowProgress**, and therefore our use of our windows, shows up on the correct thread (making sure to replace both calls to **ShowProgress** in **CalcPi**):

```
delegate
void ShowProgressDelegate(string pi, int totalDigits, int digitsSoFar);

void CalcPi(int digits) {
  StringBuilder pi = new StringBuilder("3", digits + 2);

  // Get ready to show progress asynchronously
  ShowProgressDelegate showProgress =
    new ShowProgressDelegate(ShowProgress);

  // Show progress
  this.Invoke(showProgress, new object[] { pi.ToString(), digits, 0});

  if( digits > 0 ) {
    pi.Append(".");

    for( int i = 0; i < digits; i += 9 ) {
      ...
      // Show progress
      this.Invoke(showProgress,
        new object[] { pi.ToString(), digits, i + digitCount});
    }
  }
}
```

The use of **Invoke** has finally given me a safe use of multithreading in my Windows Forms application. The UI thread spawns a worker thread to do the long-running operation, and the worker thread passes control back to the UI thread when the UI needs updating. Figure 5 shows our safe multithreading architecture.

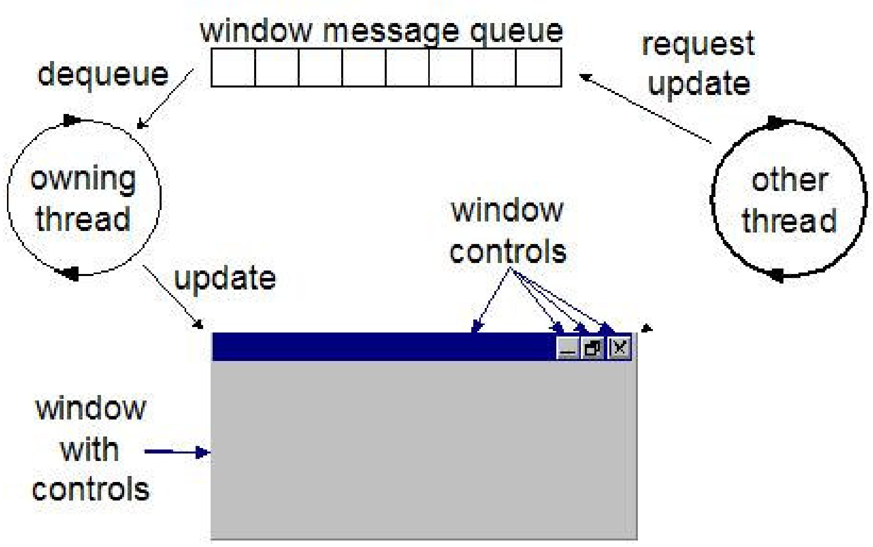


Figure 5. Safe multithreading

Simplified Multithreading

The call to **Invoke** is a bit cumbersome, and because it happens twice in our **CalcPi** function, we could simplify things and update **ShowProgress** itself to do the asynchronous call. If **ShowProgress** is called from the correct thread, it will update the controls, but if it's called from the incorrect thread, it uses **Invoke** to call itself back on the correct thread. This lets us go back to the previous, simpler **CalcPi**:

```
void ShowProgress(string pi, int totalDigits, int digitsSoFar) {
  // Make sure we're on the right thread
  if( _pi.InvokeRequired == false ) {
    _pi.Text = pi;
    _piProgress.Maximum = totalDigits;
    _piProgress.Value = digitsSoFar;
  }
  else {
    // Show progress asynchronously
    ShowProgressDelegate showProgress =
      new ShowProgressDelegate(ShowProgress);
    this.Invoke(showProgress,
      new object[] { pi, totalDigits, digitsSoFar});
  }
}

void CalcPi(int digits) {
  StringBuilder pi = new StringBuilder("3", digits + 2);

  // Show progress
  ShowProgress(pi.ToString(), digits, 0);

  if( digits > 0 ) {
    pi.Append(".");
```

```
      for( int i = 0; i < digits; i += 9 ) {
        ...
        // Show progress
        ShowProgress(pi.ToString(), digits, i + digitCount);
      }
    }
  }
```

Because **Invoke** is a synchronous call and we're not consuming the return value (in fact, **ShowProgress** doesn't have a return value), it's better to use **BeginInvoke** here so that the worker thread isn't held up, as shown here:

```
BeginInvoke(showProgress, new object[] { pi, totalDigits, digitsSoFar});
```

BeginInvoke is always preferred if you don't need the return of a function call because it sends the worker thread to its work immediately and avoids the possibility of deadlock.

Where Are We?

I've used this short example to demonstrate how to perform long-running operations while still showing progress and keeping the UI responsive to user interaction. To accomplish this, I used one asynch delegate to spawn a worker thread and the **Invoke** method on the main form, along with another delegate to be executed back in the UI thread.

One thing I was very careful never to do was to share access to a single point of data between the UI thread and the worker thread. Instead, I passed a copy of the data needed to do the work to the worker thread (the number of digits), and a copy of the data needed to update the UI (the digits calculated so far and the progress). In the final solution, I never passed references to objects that I was sharing between the two threads, such as a reference to the current StringBuilder (which would have saved me a string copy for every time I went back to the UI thread). If I had passed shared references back and forth, I would have had to use .NET synchronization primitives to make sure to that only one thread had access to any one object at a time, which would have been a lot of work. It was already enough work just to get the calls happening between the two threads without bringing synchronization into it.

Of course, if you've got large datasets that you're working with you're not going to want to copy data around. However, when possible, I recommend the combination of asynchronous delegates and message passing between the worker thread and the UI thread for implementing long-running tasks in your Windows Forms applications.

Acknowledgments

I'd like to thank Simon Robinson for his post on the DevelopMentor .NET mailing list that inspired this article, Ian Griffiths for his initial work in this area, Chris Andersen for his message-passing ideas, and last but certainly not least, Mike Woodring for the fabulous multithreading pictures that I lifted shamelessly for this article.

References

- This article's source code
- .NET Delegates: A C# Bedtime Story
- *Win32 Multithreaded Programming* by Mike Woodring and Aaron Cohen

---

Chris Sells is an independent consultant, specializing in distributed applications in .NET and COM, as well as an instructor for DevelopMentor. He's written several books, including *ATL Internals*, which is in the process of being updated for ATL7. He's also working on *Essential Windows Forms* for Addison-Wesley and *Mastering Visual Studio .NET* for O'Reilly. In his free time, Chris hosts the Web Services DevCon and directs the Genghis source-available project. More information about Chris, and his various projects, is available at http://www.sellsbrothers.com.

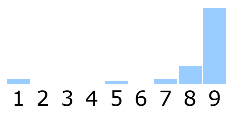
Print E-Mail

How would you rate the quality of this content?

	1	2	3	4	5	6	7	8	9	
Poor	○	○	○	○	○	○	○	○	○	Outstanding

Tell us why you rated the content this way. (optional)

Average rating:
8 out of 9



395 people have rated this page

Safe, Simple Multithreading in Windows Forms, Part 2

Chris Sells

September 2, 2002

Summary: Explores how to leverage multiple threads to split the user interface (UI) from a long-running operation while communicating further user input to the worker thread to adjust its behavior, thus allowing a message-passing scheme for robust, correct multithreaded processing. (8 printed pages)

Download the asynchcaclpi.exe sample file.

As you may recall from a couple of columns ago, Safe, Simple Multithreading in Windows Forms, Part 1, Windows Forms and threading can be used together with good results if you're careful. Threading is a nice way to perform long-running operations, like calculating pi to a large number of digits as shown in Figure 1 below.

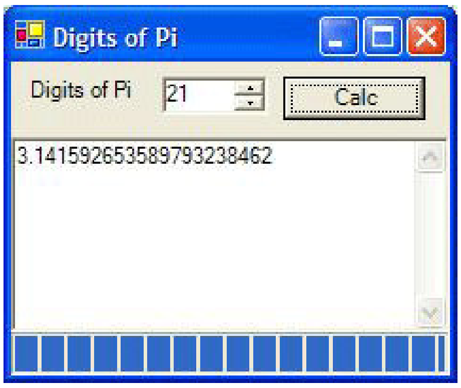


Figure 1. Digits of Pi application

Windows Forms and Background Processing

In the last article, we explored starting threads directly for background processing, but settled on using asynchronous delegates to get our worker thread started. Asynchronous delegates have the convenience of syntax when passing arguments and scale better by taking threads from a process-wide, common language runtime-managed pool. The only real problem we ran into was when the worker thread wanted to notify the user of progress. In this case, it wasn't allowed to work with the UI controls directly (a long-standing Win32® UI no-no). Instead, the worker thread has to send or post a message to the UI thread, using **Control.Invoke** or **Control.BeginInvoke** to cause code to execute on the thread that owns the controls. These considerations resulted in the following code:

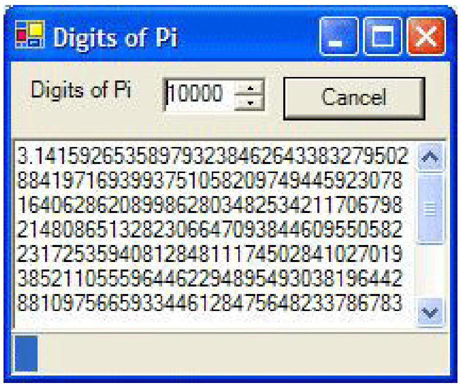
```
// Delegate to begin asynch calculation of pi
delegate void CalcPiDelegate(int digits);
void _calcButton_Click(object sender, EventArgs e) {
  // Begin asynch calculation of pi
  CalcPiDelegate calcPi = new CalcPiDelegate(CalcPi);
  calcPi.BeginInvoke((int)_digits.Value, null, null);
}

void CalcPi(int digits) {
  StringBuilder pi = new StringBuilder("3", digits + 2);

  // Show progress
  ShowProgress(pi.ToString(), digits, 0);

  if( digits > 0 ) {
    pi.Append(".");

    for( int i = 0; i < digits; i += 9 ) {
      ...
      // Show progress
      ShowProgress(pi.ToString(), digits, i + digitCount);
    }
  }
}

// Delegate to notify UI thread of worker thread progress
delegate
void ShowProgressDelegate(string pi, int totalDigits, int digitsSoFar);

void ShowProgress(string pi, int totalDigits, int digitsSoFar) {
  // Make sure we're on the right thread
  if( _pi.InvokeRequired == false ) {
    _pi.Text = pi;
    _piProgress.Maximum = totalDigits;
    _piProgress.Value = digitsSoFar;
  }
  else {
    // Show progress synchronously
    ShowProgressDelegate showProgress =
      new ShowProgressDelegate(ShowProgress);
    this.BeginInvoke(showProgress,
      new object[] { pi, totalDigits, digitsSoFar });
  }
}
```

Notice that we have two delegates. The first, **CalcPiDelegate**, is for use in bundling up the arguments to be passed to **CalcPi** on the worker thread allocated from the thread pool. An instance of this delegate is created in the event handler when the user decides they wants to calculate pi. The work is queued to the thread pool by calling **BeginInvoke**. This first delegate is really for the UI thread to pass a message to the worker thread.

The second delegate, **ShowProgressDelegate**, is for use by the worker thread that wants to pass a message back to the UI thread, specifically an update on how things are progressing in the long-running operation. To shield callers from the details of thread-safe communication with the UI thread, the **ShowProgress** method uses the **ShowProgressDelegate** to send a message to itself on the UI thread through the **Control.BeginInvoke** method. **Control.BeginInvoke** asynchronously queues work up for the UI thread and then continues on without waiting for the result.

Canceling

In this example, we're able to send messages back and forth between the worker and the UI threads without a care in the world. The UI thread doesn't have to wait for the worker thread to complete or even be notified on completion because the worker thread communicates it progress as it goes. Likewise, the worker thread doesn't have to wait for the UI thread to show progress so long as progress messages are sent at regular intervals to keep users happy. However, one thing doesn't make users happy—not having full control of any processing that their applications are performing. Even though the UI is responsive while pi is being calculated, the user would still like the option to cancel the calculation if they've decided they need 1,000,001 digits and they mistakenly asked for only 1,000,000. An updated UI for **CalcPi** that allows for cancellation is shown in Figure 2.



```
3.1415926535897932384626433832795028
8419716939937510582097494459230781
6406286208998628034825342117067982
1480865132823066470938446095505822
3172535940812848111745028410270193
8521105559644622948954930381964428
8109756659334461284756482337867831
```

Figure 2. Letting the user cancel a long-running operation

Implementing cancel for a long running operation is a multi-step process. First, a UI needs to be provided for the user. In this case, the **Calc** button was changed to a **Cancel** button after a calculation has begun. Another popular choice is a progress dialog, which typically includes current progress details, including a progress bar showing percentage of work complete, as well as a **Cancel** button.

If the user decides to cancel, that should be noted in a member variable and the UI disabled for the small amount of time between when the UI thread knows the worker thread should stop, but before the worker thread itself knows and has a chance to stop sending progress. If this period of time is ignored, it's possible that the user could start another operation before the first worker thread stops sending progress, making it the job of the UI thread to figure out whether it's getting progress from the new worker thread or the old worker thread that's supposed to be shutting down. While it's certainly possible to assign each worker thread a unique ID so that the UI thread can keep such things organized (and, in the face of multiple simultaneous long-running operations, you may well need to do this), it's often simpler to pause the UI for the brief amount of time between when the UI knows the worker thread is going to stop, but before the worker thread knows. Implementing it for our simple pi calculator is a matter of keeping a tri-value enum, as shown here:

```
enum CalcState {
    Pending,     // No calculation running or canceling
    Calculating, // Calculation in progress
    Canceled,    // Calculation canceled in UI but not worker
}

CalcState _state = CalcState.Pending;
```

Now, depending on what state we're in, we treat the **Calc** button differently, as shown here:

```
void _calcButton_Click(...)  {
    // Calc button does double duty as Cancel button
    switch( _state ) {
        // Start a new calculation
        case CalcState.Pending:
            // Allow canceling
            _state = CalcState.Calculating;
            _calcButton.Text = "Cancel";

            // Asynch delegate method
            CalcPiDelegate  calcPi = new CalcPiDelegate(CalcPi);
            calcPi.BeginInvoke((int)_digits.Value, null, null);
            break;

        // Cancel a running calculation
        case CalcState.Calculating:
            _state = CalcState.Canceled;
            _calcButton.Enabled = false;
            break;

        // Shouldn't be able to press Calc button while it's canceling
        case CalcState.Canceled:
            Debug.Assert(false);
            break;
    }
}
```

Notice that when the **Calc**/**Cancel** button is pressed in `Pending` state, we send the state to `Calculating` (as well as changing the label on the button), and start the calculation asynchronously as we did before. If the state is `Calculating` when the **Calc**/**Cancel** button is pressed, we switch the state to `Canceled` and disable the UI to start a new calculation for as long as it takes us to communicate the canceled stated to the worker thread. Once we've communicated to the worker thread that the operation should be canceled, we'll enable the UI again and reset the state back to `Pending` so that the user can begin another operation. To communicate to the worker that it should cancel, let's augment the **ShowProgress** method to include a new *out* parameter:

```
void ShowProgress(..., out bool cancel)

void CalcPi(int digits) {
    bool cancel = false;
    ...

    for( int i = 0; i < digits; i += 9 ) {
        ...

        // Show progress (checking for Cancel)
        ShowProgress(..., out cancel);
        if( cancel ) break;
    }
}
```

You may be tempted to make the cancel indicator a Boolean return value from **ShowProgress**, but I can never remember if `true` means to cancel or that everything is fine/continue as normal, so I use the *out* parameter technique, which even I can keep straight.

The only thing left to do is updating the **ShowProgress** method, which is the code that actually performs the transition between worker thread and UI thread, to notice if the user has asked to cancel and to let **CalcPi** know accordingly. Exactly how we communicate that information between the UI and the worker thread depends on which technique we'd like to use.

Communication with Shared Data

The obvious way to communicate the current state of the UI is to let the worker thread access the *_state* member variable directly. We could accomplish this with the following code:

```
void ShowProgress(..., out bool cancel) {
  // Don't do this!
  if( _state == CalcState.Cancel ) {
    _state = CalcState.Pending;
    cancel = true;
  }
  ...
}
```

I hope that something inside you cringed when you saw this code (and not just because of the warning comment). If you're going to be doing multithreaded programming, you're going to have to watch out any time that two threads can have simultaneous access to the same data (in this case, the *_state* member variable). Shared access to data between threads makes it easy to get into *race conditions* where one thread is racing to read data that is only partially up-to-date before another thread has finished updating it. For concurrent access to shared data to work, you need to monitor usage of your shared data to make sure that each thread waits patiently while the other thread works on the data. To monitor access to shared data, .NET provides the **Monitor** class to be used on a shared object to act as the lock on the data, which C# wraps with the handy lock block:

```
object _stateLock = new object();

void ShowProgress(..., out bool cancel) {
  // Don't do this either!
  lock( _stateLock ) { // Monitor the lock
    if( _state == CalcState.Cancel ) {
      _state = CalcState.Pending;
      cancel = true;
    }
    ...
  }
}
```

Now I've properly locked access to the shared data, but I've done it in such a way as to make it very likely to cause another common problem when performing multithreaded programming—*deadlock*. When two threads are deadlocked, both of them wait for the other to complete their work before continuing, making sure that neither actually progresses.

If all this talk of race conditions and deadlocks has caused you concern—good. Multithreaded programming with shared data is darn hard. So far we've been able to avoid these issues because we have been passing around copies of data of which each thread gets complete ownership. Without shared data, there's no need for synchronization. If you find that you need access to shared data (that is, the overhead of copying the data is too great a space or time burden), then you'll need to study up on sharing data between threads (check the References section for my favorite study aid in this area).

However, the vast majority of multithreading scenarios, especially as related to UI multithreading, seem to work best with the simple message-passing scheme we've been using so far. Most of the time, you don't want the UI to have access to data being worked on in the background (the document being printed or the collection of objects being enumerated, for example). For these cases, avoiding shared data is the best choice.

Communicating with Method Parameters

We've already augmented our **ShowProgress** method to contain an *out* parameter. Why not let **ShowProgress** check the state of the *_state* variable when it's executing on the UI thread, like so:

```
void ShowProgress(..., out bool cancel) {
    // Make sure we're on the UI thread
    if( _pi.InvokeRequired == false ) {
        ...

        // Check for Cancel
        cancel = (_state == CalcState.Canceled);

        // Check for completion
        if( cancel || (digitsSoFar == totalDigits) ) {
            _state = CalcState.Pending;
            _calcButton.Text = "Calc";
            _calcButton.Enabled = true;

        }
    }
    // Transfer control to UI thread
    else { ... }
}
```

Because the UI thread is the only one accessing the *_state* member variable, no synchronization is needed. Now it's just a matter of passing control to the UI thread in such a way as to harvest the *cancel out* parameter of the **ShowProgressDelegate**. Unfortunately, our use of **Control.BeginInvoke** makes this complicated. The problem is that **BeginInvoke** won't wait around for a result of calling **ShowProgress** on the UI thread, so we have two choices. One option is to pass another delegate to **BeginInvoke** to be called when **ShowProgress** has returned from the UI thread, but that will happen on another thread in the thread pool, so we'll have to go back to synchronization, this time between the worker thread another thread from the pool. A simpler option is to switch to the synchronous **Control.Invoke** method and wait for the *cancel out* parameter. However, even this is a bit tricky as you can see in the following code:

```
void ShowProgress(..., out bool cancel) {
    if( _pi.InvokeRequired == false ) { ... }
    // Transfer control to UI thread
    else {
        ShowProgressDelegate  showProgress =
            new ShowProgressDelegate(ShowProgress);

        // Avoid boxing and losing our return value
        object inoutCancel = false;

        // Show progress synchronously (so we can check for cancel)
        Invoke(showProgress, new object[] { ..., inoutCancel});
        cancel = (bool)inoutCancel;
    }
}
```

While it would have been ideal to simply pass a Boolean variable directly to **Control.Invoke** to harvest the cancel parameter, we have a problem. The problem is that *bool* is a *value data type*, whereas **Invoke** takes an array of objects as parameters, and objects are *reference data types*. Check the References section for books discussing the difference, but the upshot is that a *bool* passed as an object will be copied, leaving our actual *bool* unchanged, meaning we'd never know when the operation was canceled. To avoid this situation, we create our own object variable (*inoutCancel*) and pass it instead, avoiding the copy. After the synchronous call to **Invoke**, we cast the *object* variable to a *bool* to see if the operation should be canceled or not.

The value versus reference type distinction is something you'll have to watch out for whenever you call **Control.Invoke** (or **Control.BeginInvoke**) with *out* or *ref* parameters that are value types, such as primitive types like *int* or *bool*, as well as enums and structs. However, if you've got more complicated data being passed around as a custom reference type aka class, you don't need to do anything special to make things work. However, even the unpleasantness of handling value types with **Invoke**/**BeginInvoke** pales in comparison to getting multithreaded code to access shared data in a race condition/deadlock-free way, so I consider it a small price to pay.

Conclusion

Once again, we've used a seemingly trivial example to explore some complicated issues. Not only have we leveraged multiple threads to split the UI from a long-running operation, but also we've communicated further user input back to the worker thread to adjust its behavior. While we could have used shared data, to avoid the complications of synchronization (which only arise when your boss tries your code), we've stayed with our message-passing scheme for robust, correct multithreaded processing.

References

- This article's source code
- Safe, Simple Multithreading in Windows Forms, Part 1
- Win32 Multithreaded Programming by Aaron Cohen and Mike Woodring
- Applied Microsoft .NET Framework Programming by Jeffrey Richter
- Essential .NET, Volume 1: The Common Language Runtime by Don Box

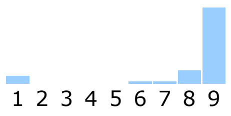
Print E-Mail

How would you rate the quality of this content?

	1	2	3	4	5	6	7	8	9	
Poor	○	○	○	○	○	○	○	○	○	Outstanding

Tell us why you rated the content this way. (optional)

Average rating:
8 out of 9



123 people have rated this page

Manage Your Profile | Legal | Contact Us | MSDN Flash Newsletter

Safe, Simple Multithreading in Windows Forms, Part 3

Page Options

Average rating:
7 out of 9

Rate this page

Print this page

E-mail this page

Chris Sells
Sells Brothers Consulting

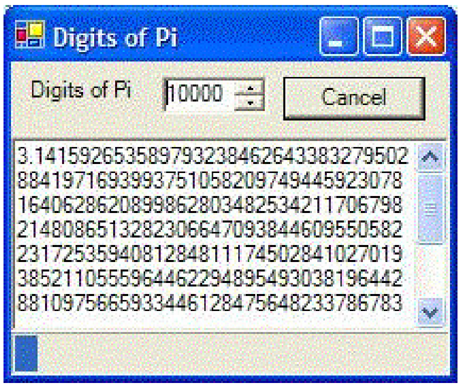
January 23, 2003

Summary: Chris Sells discusses how to more cleanly communicate between a UI thread and a worker thread, and how to call Web services asynchronously in a Windows Forms application. (11 printed pages)

Download the winforms01232003.exe sample file.

I was once described in a conference talk as "pedantic." Since taking the *Webster's Dictionary* definition to heart can only lead to hurt feelings, I prefer my friend's definition, who says that I'm "like a dog with a bone." This latter definition I take as a complement because I do like to make sure that I do a job all the way. In this case, I found that I still had more to say on the topic of Windows Forms and multithreading, both as to how to more cleanly communicate between a UI thread and a worker thread, and how to call Web services asynchronously in a Windows Forms application.

Communication with Message Passing



```
3.1415926535897932384626433832795028
884197169399375105820974944592307816
406286208998628034825342117067982148
086513282306647093844609550582231725
359408128481117450284102701938521105
559644622948954930381964428810975665
9334461284756482337867831
```

Figure 1. Calculating pi

Recall from Safe, Simple Multithreading in Windows Forms, Part 2 that we had built a client for calculating pi to an arbitrary number of digits on a worker thread, leaving us with the following function to communicate results from the worker thread to the UI thread:

```
void ShowProgress(..., out bool cancel) {
    if( _pi.InvokeRequired == false ) { ... }
    // Transfer control to UI thread
    else {
        ShowProgressDelegate  showProgress =
            new ShowProgressDelegate(ShowProgress);

        // Avoid boxing and losing our return value
        object inoutCancel = false;

        // Show progress synchronously (so we can check for cancel)
        Invoke(showProgress, new object[] { ..., inoutCancel});
        cancel = (bool)inoutCancel;
    }
}
```

Remember that we were using an out parameter to communicate back from the UI thread, regardless of whether the user had canceled the operation or not. Because the out parameter was a value type instead of a reference type, we had to write some pretty unpleasant code to avoid updating a boxed copy in the UI thread and losing that information in the worker thread. The simplicity of the **CalcPi** example, and the resulting complexity of sending around a single Boolean indicating whether to cancel or not, may cause us to try a solution like the following:

```
void ShowProgress(..., out bool cancel) {
  // Make sure we're on the right thread
  if( this.InvokeRequired == false ) {...}
  // Transfer control to correct thread
  else {
    ShowProgressDelegate
      showProgress = new ShowProgressDelegate(ShowProgress);

    // Show progress synchronously (so we can check for cancel)
    Invoke(showProgress, new object[] {...});

    // Check for Cancel the easy, but special-purpose, way
    cancel = (state == CalcState.Canceled);
  }
}
```

For our simple application, and others like it, that would work just fine. Since the worker thread only reads from the state field, it will always be valid (although a race condition could cause it to be out of date). However, don't be tempted down this path. As soon as we have multiple outstanding requests and we keep them in an array or any kind of data structure, we run the risk of attempting to access data that has been invalidated (those darn race conditions again), which we'll have to protect against using synchronization (remember deadlocks?). It's so much simpler and safer to pass around ownership of the data instead of sharing access to the same data. If the weird boxing thing required by the Control.Invoke call turns you off, I recommend following the delegate pattern used by the rest of the .NET Framework:

```
class ShowProgressArgs : EventArgs {
  public string Pi;
  public int TotalDigits;
  public int DigitsSoFar;
  public bool Cancel;

  public ShowProgressArgs(
    string pi, int totalDigits, int digitsSoFar) {
    this.Pi = pi;
    this.TotalDigits = totalDigits;
    this.DigitsSoFar = digitsSoFar;
  }
}

delegate void ShowProgressHandler(object sender, ShowProgressArgs e);
```

This code declares the class **ShowProgressArgs** that derives from the **EventArgs** base class to hold event arguments, as well as a delegate that takes a sender and an instance on the custom arguments object. With this in place, we can change the **ShowProgress** implementation like so:

```
void ShowProgress(...) {
  // Make sure we're on the right thread
  if( this.InvokeRequired == false ) {...}
  }
  // Transfer control to correct thread
  else {
    ShowProgressHandler
      showProgress =
        new ShowProgressHandler(AsynchCalcPiForm_ShowProgress);
    object sender = System.Threading.Thread.CurrentThread;
    ShowProgressArgs
      e = new ShowProgressArgs(pi, totalDigits, digitsSoFar);
    Invoke(showProgress, new object[] {sender, e});
    cancel = e.Cancel;
  }
}

void AsynchCalcPiForm_ShowProgress(object sender, ShowProgressArgs e) {
  ShowProgress(e.Pi, e.TotalDigits, e.DigitsSoFar, out e.Cancel);
}
```

ShowProgress hasn't changed in signature, so **CalcPi** still calls it in the same simple way, but now the worker thread will compose an instance of the **ShowProgressArgs** object to pass the appropriate arguments to the UI thread through a handler that looks like any other event handler, including a sender and an **EventArgs**-derived object. It turns around and calls the **ShowProgress** method again, breaking out the arguments from the **ShowProgressArgs** object. After **Invoke** returns in the worker thread, it pulls out the cancel flag without any concern about boxing because the **ShowProgressArgs** type is a reference type. However, even though it is a reference type and the worker thread passes control of it to the UI thread, there's no danger of race conditions because the worker thread waits 'til the UI thread is done working with the data before accessing it again.

This usage could be further simplified if we want the **CalcPi** function to create instances of **ShowProgressArgs** itself, eliminating the need for an intermediate method:

```
void ShowProgress(object sender, ShowProgressArgs e) {
  // Make sure we're on the right thread
  if( this.InvokeRequired == false ) {
    piTextBox.Text = e.Pi;
    piProgressBar.Maximum = e.TotalDigits;
    piProgressBar.Value = e.DigitsSoFar;

    // Check for Cancel
    e.Cancel = (state == CalcState.Canceled);

    // Check for completion
    if( e.Cancel || (e.DigitsSoFar == e.TotalDigits) ) {
      state = CalcState.Pending;
      calcButton.Text = "Calc";
      calcButton.Enabled = true;

    }
  }
  // Transfer control to correct thread
  else {
    ShowProgressHandler
      showProgress =
      new ShowProgressHandler(ShowProgress);
    Invoke(showProgress, new object[] { sender, e});
  }
}

void CalcPi(int digits) {
  StringBuilder pi = new StringBuilder("3", digits + 2);
  object sender = System.Threading.Thread.CurrentThread;
  ShowProgressArgs e = new ShowProgressArgs(pi.ToString(), digits, 0);

  // Show progress (ignoring Cancel so soon)
  ShowProgress(sender, e);

  if( digits > 0 ) {
    pi.Append(".");

    for( int i = 0; i < digits; i += 9 ) {
      int nineDigits = NineDigitsOfPi.StartingAt(i+1);
      int digitCount = Math.Min(digits - i, 9);
      string ds = string.Format("{0:D9}", nineDigits);
      pi.Append(ds.Substring(0, digitCount));

      // Show progress (checking for Cancel)
      e.Pi = pi.ToString();
      e.DigitsSoFar = i + digitCount;
      ShowProgress(sender, e);
      if( e.Cancel ) break;
    }
  }
}
```

This technique represents a *message passing* model. This model is clear, safe, and general-purpose. It's clear because it's easy to see that the worker is creating a message, passing it to the UI, and then checking the message for extra information during the UI thread's processing of the message. It's safe because the ownership of the message is never shared, starting with the worker thread, moving to the UI thread, and then returning to the worker thread with no simultaneous access between the two threads. It's general-purpose because if the worker or UI thread needed to communicate additional information besides a cancel flag, that information can be added to the **ShowProgressArgs** class. Message passing is what our applications should be using to communicate between UI and working threads.

Asynchronous Web Services

One specific area in which we'll want to use Windows Forms applications in an asynchronous manner is Web services. Web services are like passing messages between threads except that the messages travel between machines using standard protocols, such as HTTP and XML. Imagine a .NET Framework-based Web service that calculated digits of pi using some way-cool, fast pi calculation engine:

```
public class CalcPiService : System.Web.Services.WebService {
  [WebMethod]
  public string CalcPi(int digits) {
    StringBuilder pi = new StringBuilder("3", digits + 2);

    // Way cool fast pi calculator running on a huge processor...

    return pi.ToString();
  }
}
```

Now imagine a version of the **CalcPi** program that used the Web service instead of our slow client-side algorithm to calculate pi on giant machines with huge processors (or even better, databases with more digits of pi cached than anyone could ever want or need). Although the underlying protocols of Web services are HTTP- and XML-based, and we could form a Web service request fairly easily to ask for the digits of pi we're after, it's even simpler to use the client-side Web services proxy generation built into Visual Studio® .NET in the **Project** menu with the **Add Web Reference** item. The resultant **Add Web Service** dialog allows us to enter the URL of the WSDL (Web Service Description Language) that describes the Web service we'd like to call; http://localhost/CalcPiWebService/CalcPiService.asmx?WSDL for example. In this case some code will be added to our project from http://localhost/CalcPiWebService/CalcPiService.asmx?WSDL. To see it, we'll need to look in the Solution Explorer under the <SolutionName>/<ProjectName>/Web References/<ServerName>/Reference.map/Reference.cs. The generated proxy code for the **CalcPi** Web service looks like this:

```
namespace AsynchCalcPi.localhost {
  [WebServiceBindingAttribute(Name="CalcPiServiceSoap", ...)]
    public class CalcPiService : SoapHttpClientProtocol {
    public CalcPiService() {
      this.Url =
        "http://localhost/CalcPiWebService/CalcPiService.asmx";
    }

    [SoapDocumentMethodAttribute("http://tempuri.org/CalcPi", ...)]
    public string CalcPi(int digits) {...}

    public IAsyncResult
      BeginCalcPi(
      int digits,
      System.AsyncCallback callback,
      object asyncState) {...}

    public string EndCalcPi(IAsyncResult asyncResult) {...}
  }
}
```

Calling the Web service is now a matter of creating and instance of the proxy and calling the method we're interested in:

```
localhost.CalcPiService service = new localhost.CalcPiService();

void calcButton_Click(object sender, System.EventArgs e) {
  piTextBox.Text = service.CalcPi((int)digitsUpDown.Value);
}
```

As with other kinds of long-running methods, if this one takes a long time it'll block the UI thread. The standard techniques already discussed in this series can be used to call synchronous Web service methods asynchronously, but as you can tell in the generated proxy code, there's already built-in support for asynchronous operations through the BeginXxx/EndXxx method pairs, one for each method on the Web service.

Retrofitting our existing **CalcPi** client to use the Web service begins by calling the Web service's **CalcPi** instead of our own:

```
enum CalcState {
  Pending,
  Calculating,
  Canceled,
}

CalcState state = CalcState.Pending;
localhost.CalcPiService service = new localhost.CalcPiService();

void calcButton_Click(object sender, System.EventArgs e) {
  switch( state ) {
    case CalcState.Pending:
      state = CalcState.Calculating;
      calcButton.Text = "Cancel";

      // Start web service request
      service.BeginCalcPi(
        (int)digitsUpDown.Value,
        new AsyncCallback(PiCalculated),
        null);
      break;

    case CalcState.Calculating:
      state = CalcState.Canceled;
      calcButton.Enabled = false;
      service.Abort(); // Fail all outstanding requests
      break;

    case CalcState.Canceled:
      Debug.Assert(false);
      break;
  }
}

void PiCalculated(IAsyncResult res) {...}
```

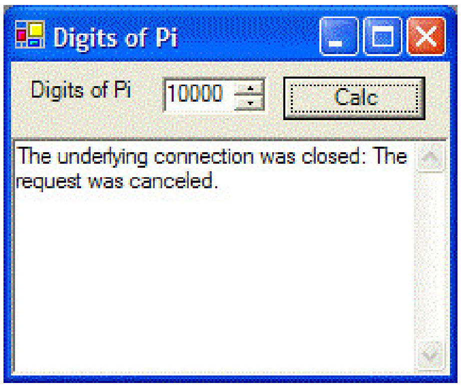
This time, instead of declaring a custom delegate to start the asynchronous operation, **BeginCalcPi** takes, in addition to the method parameters, an instance of an **AsyncCallback** delegate, which returns void and takes an **IAsyncResult** as the sole argument. The application provides the **PiCalculated** method that matches that signature. The **PiCalculated** method will be called when the Web service returns and is responsible for harvesting results.

Also, while there is no way to get progress from a Web service (all the more reason to call it asynchronously), calls to a Web service can be canceled by calling the **Abort** method. Be careful with this one, however, as it will cancel all outstanding requests, not just a specific one.

When the **PiCalculated** method is called, that means that the Web service has returned something:

```
void PiCalculated(IAsyncResult res) {
  try {
    ShowPi(service.EndCalcPi(res));
  }
  catch( WebException ex ) {
    ShowPi(ex.Message);
  }
}
```

The call to **EndCalcPi**, passing in the **IAsyncResult** parameter, provides whatever results we've gotten by calling the Web service. If the Web service call was successful, pi is the returned value. If, on the other hand, the Web service call was unsuccessful because it timed out or was canceled, **EndCalcPi** will thrown a WebException error, which is why **PiCalculated** wraps the call to **EndCalcPi** in a try-catch block. **ShowPi** will be called on to either show the digits of pi that were calculated by the Web service or the exception message (the result of a canceled Web service call is shown in Figure 2).



The underlying connection was closed: The request was canceled.

Figure 2. The result of a canceled call to the CalcPi Web service

```
delegate void ShowPiDelegate(string pi);

void ShowPi(string pi) {
  if( this.InvokeRequired == false ) {
    piTextBox.Text = pi;
    state = CalcState.Pending;
    calcButton.Text = "Calc";
    calcButton.Enabled = true;
  }
  else {
    ShowPiDelegate showPi = new ShowPiDelegate(ShowPi);
    this.BeginInvoke(showPi, new object[] {pi});
  }
}
```

Notice that the **ShowPi** method contains the check to see if an invoke is required and makes a call to **BeginInvoke** to transfer from the current thread to the UI thread. This is necessary because the Web service completion notification will come in on some thread other than the UI thread, and by now we know how important it is to translate the call back to the UI thread before touching any of the controls.

Where Are We?

Just when you thought it was safe to eat pie again, I had more things to say about how to handle multithreaded operations in Windows Forms cleanly and safely. Not only does this simple example leverage multiple threads to split the UI from a long-running operation, but also the UI thread communicates further user input back to the worker thread to adjust its behavior. While it could have used shared data, the application used a message-passing scheme between threads to avoid the complications of synchronization. Finally, the application was further extended to take advantage of asynchronous calls to Web services, which is really the only way to call them as even quick methods can take arbitrary long amounts of time across the Web.

> **Note** This material is excerpted from the forthcoming Addison-Wesley title: *Windows Forms Programming in C#* by Chris Sells (0321116208). Please note the material presented here is an initial DRAFT of what will appear in the published book.

Chris Sells in an independent consultant, speaker and author specializing in distributed applications in .NET and COM. He's written several books and is currently working on *Windows Forms for C# and VB.NET Programmers* and *Mastering Visual Studio .NET*. In his free time, Chris hosts various conferences, directs the Genghis source-available project, plays with Rotor and, in general, makes a pest of himself at Microsoft design reviews. More information about Chris, and his various projects, is available at http://www.sellsbrothers.com.

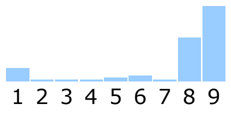
Print E-Mail

How would you rate the quality of this content?

	1	2	3	4	5	6	7	8	9	
Poor	○	○	○	○	○	○	○	○	○	Outstanding

Tell us why you rated the content this way. (optional)

Average rating:
7 out of 9



129 people have rated this page