Keep Your UI Responsive with the BackgroundWorker Component

An overview of the BackgroundWorker component by JeremyBytes.com

The Problem

We've all experienced it: the application UI that hangs. You get the dreaded "Not Responding" message, and you have to decide if you should wait it out or simply kill the process. If you have long-running processes in your application, you should consider putting them on a separate thread so that your UI remains responsive. However, threading is a daunting subject. We've heard horror stories about race conditions and deadlocks, and needing to use the thread dispatcher to communicate between background threads and the UI thread. At this point, it sounds like a subject best left to the experts.

The Solution

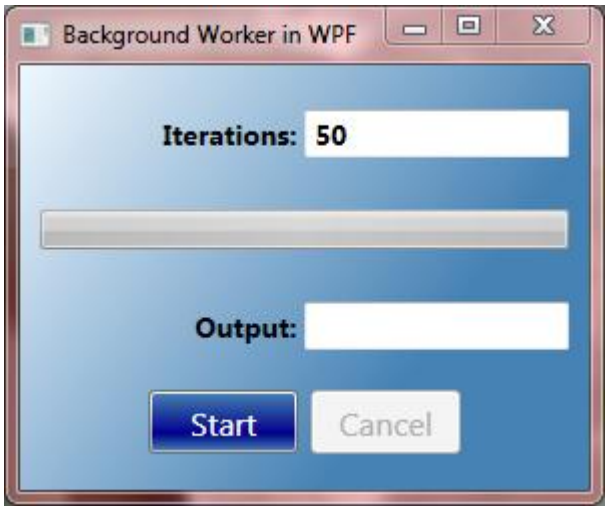
Fortunately, the .NET framework provides a simple way to get started in threading with the BackgroundWorker component. This wraps much of the complexity and makes spawning a background thread relatively safe. In addition, it allows you to communicate between your background thread and your UI thread without doing any special coding. You can use this component with WinForms, WPF and Silverlight applications. We'll be using it with WPF here.

The BackgroundWorker offers several features which include spawning a background thread, the ability to cancel the background process before it has completed, and the chance to report the progress back to your UI. We'll be looking at all of these features.

The Set Up

We'll start with a fairly simple WPF application that has a long-running process that blocks the application until it has completed. You can download the source code for the application here: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at here is built for Visual Studio 2010, but there is also a version for Visual Studio 2008 available for download from the same location. The download includes the starter application and the completed code. The starter application includes the following.

1. A Simple WPF form:



You can find the XAML for this in the download. It consists of 2 Text Boxes (Iterations and Output), a Progress Bar, and 2 Buttons (Start and Cancel).

2. A long running process (in the code-behind the form):

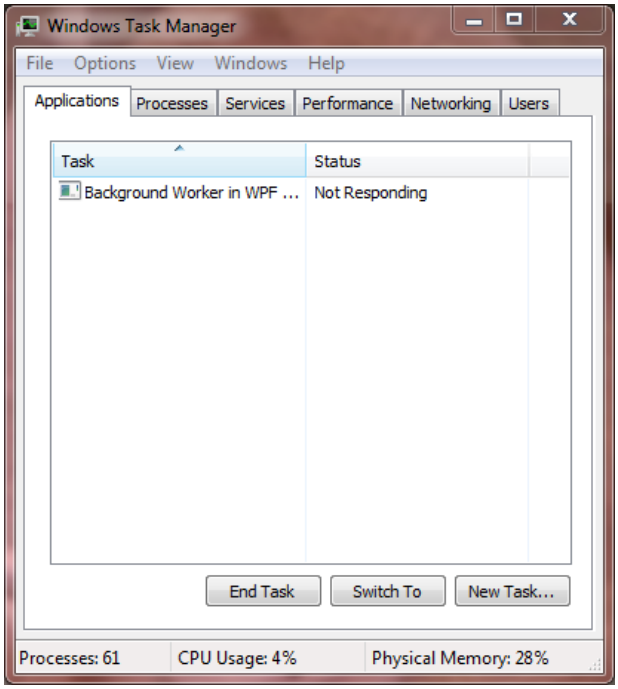
```
private int DoSlowProcess(int iterations)
{
    int result = 0;

    for (int i = 1; i <= iterations; i++)
    {
        Thread.Sleep(100);
        result = i;
    }

    return result;
}
```

You can see that we're using the famously slow process Sleep(100) that loops based on the parameter value.

3. Event-handlers for the buttons (in the code-behind):

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    int iterations = 0;
    if (int.TryParse(inputBox.Text, out iterations))
    {
        outputBox.Text = DoSlowProcess(iterations).ToString();
        startButton.IsEnabled = true;
        cancelButton.IsEnabled = false;
    }
}
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    // TODO: Implement Cancel process
}
```

When you run the application and click the Start button, you'll see that the application hangs until the process is finished. If you try to move or resize the window while the process is running, nothing will happen for several seconds. And you'll see the "Not Responding" message if you look in Task Manager:



When the process is completed, you will see the value from the Iterations box mirrored to the Output box. This is simply a confirmation that the process completed. You can try this with different values. Since we are using a Sleep(100), a value of 50 iterations will translate into a 5 second delay; a value of 100 is 10 seconds, and so on. Just as a side note, I chose a value of 1/10th of a second for the Sleep rather than the usual 1 second so that we will have a smoother progress bar update later on.

What is the BackgroundWorker Component?

The BackgroundWorker component wraps up all of the basics that you need for threading. As we'll see, it handles offloading work to a background thread (leaving the UI thread free to continue taking user input) as well as marshalling back to the UI thread for things like progress updates. Here are the members of the BackgroundWorker that we'll be dealing with:

Methods

- RunWorkerAsync
- ReportProgress
- CancelAsync

Properties

- WorkerReportsProgress
- WorkerSupportsCancellation
- CancellationPending (read-only)
- IsBusy (read-only)

Events

- DoWork
- RunWorkerCompleted
- ProgressChanged

We'll see how these are used as we move along.

Adding the BackgroundWorker

The BackgroundWorker is a non-visual component. In the WinForms world, this would mean that we could just drag the BackgroundWorker from the Tool Box onto the Form, and it would show up as a non-visual component. In WPF, things are a little bit different. We need to add the BackgroundWorker as a resource that we can reference throughout our code. Here's the steps:

1. Add the System.ComponentModel namespace to the XAML. We do this in the markup for the Window. The good news is that Visual Studio IntelliSense helps you out quite a bit with this. We'll give the namespace a "cm" alias so we can reference it easily. Here's the Window markup with the namespace included:

```
<Window x:Class="BackgroundWorkerInWPF.WorkerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:cm="clr-namespace:System.ComponentModel;assembly=System"
    Title="Background Worker in WPF" Height="250" Width="300">
...
</Window>
```

2. Add a BackgroundWorker as a Window Resource.

```
<Window.Resources>
    <cm:BackgroundWorker x:Key="backgroundWorker" />
</Window.Resources>
```

Hooking Things Up

To use the basic functionality of the BackgroundWorker, we need to do a couple of things. First, we need to hook up two event handlers: DoWork and RunWorkerCompleted. These events are much like they sound. To kick off the background process we call the RunWorkerAsyc method of the BackgroundWorker and pass any parameters we need. This fires the DoWork event (which is where

we'll put our long-running process). The RunWorkerCompleted event fires after that process is done. At that point, we can update our UI and do clean up (if required).

So, let's put our process into the background. We'll start by creating the handlers for the events mentioned above. As a reminder, Visual Studio IntelliSense helps us out quite a bit with this. In our BackgroundWorker markup that we created above, just type "DoWork=" and you'll get the option for "<New Event Handler>". This will create the stub and give the handler a name based on our component. We'll do the same for "RunWorkerCompleted" and end up with the following XAML:

```
<Window.Resources>
   <cm:BackgroundWorker x:Key="backgroundWorker"
         DoWork="BackgroundWorker_DoWork"
         RunWorkerCompleted="BackgroundWorker_RunWorkerCompleted"/>
</Window.Resources>
```

Now we'll flip over to the code-behind and implement these handlers. Let's look at the code, then we'll talk through it. Note, in addition to the code below, I have also added a "`using System.ComponentModel;`" to make things a little less verbose:

```
using System.ComponentModel;
...

private void BackgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    e.Result = DoSlowProcess((int)e.Argument);
}

private void BackgroundWorker_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else
    {
        outputBox.Text = e.Result.ToString();
    }
    startButton.IsEnabled = true;
    cancelButton.IsEnabled = false;
}
```

First, you'll see that our DoWork event calls the DoSlowProcess method (our long-running process). You'll note that we are getting an integer argument from the DoWorkEventArgs (we'll see how this gets passed in just a minute). The e.Argument is of type Object, so we have to cast it to the integer type that our method is expecting. Next, you'll note that we're passing the result back to the DoWorkEventArgs in e.Result. This will be used in the next handler.

The RunWorkerCompleted event fires after the long process is complete. You can see that the first thing we do is check to see if an error occurred. If not, then we'll go ahead and populate the output box with the result of our method. (We'll take a deeper look at error handling in just a bit.) The e.Result here is actually the same e.Result from the DoWork event. In our case, the DoSlowProcess returns an integer that we populate into the output box. In addition, you can see that we are enabling and disabling the buttons as appropriate.

The important thing here is what you **don't** see. Notice that our completed handler is manipulating our UI elements without any use of the Dispatcher or Invoke methods that you need to do if you are handling the threading on your own. Instead, we just reference the elements on the UI thread directly. The BackgroundWorker takes care of all of the complexity on the back end.

Finally, we need to kick off the DoWork event in our Start Button handler. In order to do this, we need a reference to the BackgroundWorker component in our window. The problem is that it is simply a resource right now. The first task is to get a reference to it. We'll put this code at the top of our Window class and modify the constructor:

```
private BackgroundWorker backgroundWorker;

public WorkerWindow()
{
    InitializeComponent();
    backgroundWorker =
        (BackgroundWorker)FindResource("backgroundWorker");
}
```

In this code, you see that we create a private variable to reference the BackgroundWorker component. Then in the constructor, we pull the component out of the resources by using the FindResource method. Now we can use the component in our code.

Here's our updated code in the Start Button Click event handler:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    int iterations = 0;
    if (int.TryParse(inputBox.Text, out iterations))
    {
        backgroundWorker.RunWorkerAsync(iterations);
        startButton.IsEnabled = false;
        cancelButton.IsEnabled = true;
        outputBox.Text = "";
    }
}
```

You'll notice that instead of calling the DoSlowProcess directly, we are now calling the RunWorkerAsync method of the BackgroundWorker. This method takes an optional object parameter. In our case, we

will use this to pass the number of iterations through. This is the value that shows up in the e.Argument of the DoWork handler that we saw above.

The next thing we do is update the button states appropriately. Since the Cancel button is not yet implemented, it won't have too much effect. But we'll get to that in a bit.

Finally, note that we are clearing the output text. Remember that since we are running the process in the background, our UI still remains responsive. We want to clear out the output while the process is running, and then populate it again after the process is complete. This is handled by the RunWorkerCompleted event that we saw above.

Now we have a fully-functional application with a background process. If you run the application now, you'll notice different behavior from what we saw before. After you click the Start button, the UI remains responsive: you can move and resize the window, type in the boxes, or whatever. When the process is finished, the output box is updated.

But we're not done yet. We still need to look at the Cancel and Progress functions.

Updating Progress

Next we'll look at reporting progress back from our long-running process. One thing to keep in mind if you want to have a progress bar in your UI: you will need to come up with some type of metric for the percent complete. For example, I have used the BackgroundWorker for long-running SQL queries. In that case, I was unable to report percentage because I had no idea exactly how long the process would take. In our sample here, we can use some fairly simple math to report the percentage completed.

The BackgroundWorker has a property we need to set (WorkerReportsProgress) and an event handler (ProgressChanged). These are fairly straight forward to implement. But here's where things get a little complicated. We need to update the progress in our DoSlowProcess method. This means that we need a reference to the BackgroundWorker.

Let's start with the easy parts. First the updated XAML:

```
<Window.Resources>
    <cm:BackgroundWorker x:Key="backgroundWorker"
        DoWork="BackgroundWorker_DoWork"
        RunWorkerCompleted="BackgroundWorker_RunWorkerCompleted"
        WorkerReportsProgress="True"
        ProgressChanged="BackgroundWorker_ProgressChanged"/>
 </Window.Resources>
```

Here we just set the WorkerReportsProgress to True (the default is False) and add the stub for the ProgressChanged event handler. As we did above, we'll just let Visual Studio create a <New Event Handler> for us.

To implement the event handler, we'll just set the value of the progress bar in our UI:

```
private void BackgroundWorker_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    mainProgressBar.Value = e.ProgressPercentage;
}
```

Now we'll make updates to some of our existing code. First, the DoSlowProcess method:

```
private int DoSlowProcess(int iterations,
    BackgroundWorker worker, DoWorkEventArgs e)
{
    int result = 0;

    for (int i = 0; i <= iterations; i++)
    {
        if (worker != null)
        {
            if (worker.WorkerReportsProgress)
            {
                int percentComplete =
                    (int)((float)i / (float)iterations * 100);
                worker.ReportProgress(percentComplete);
            }
        }

        Thread.Sleep(100);
        result = i;
    }

    return result;
}
```

We've added both BackgroundWorker and DoWorkEventArgs parameters to the DoSlowProcess method. In order to update the progress, we'll add some code to each iteration of the loop. First, we check to make sure that the BackgroundWorker parameter was populated; then we check the WorkerReportsProgress property to see if the BackgroundWorker reports progress. If false, then we'll skip the code. If true, then we calculate the percentage and call the BackgroundWorker.ReportProgress method. This will fire the ProgressChanged event that we implemented above.

Now, since we've added additional parameters to DoSlowProcess, we'll need to update the method call. As a reminder, this was in the DoWork event. Here's the updated code:

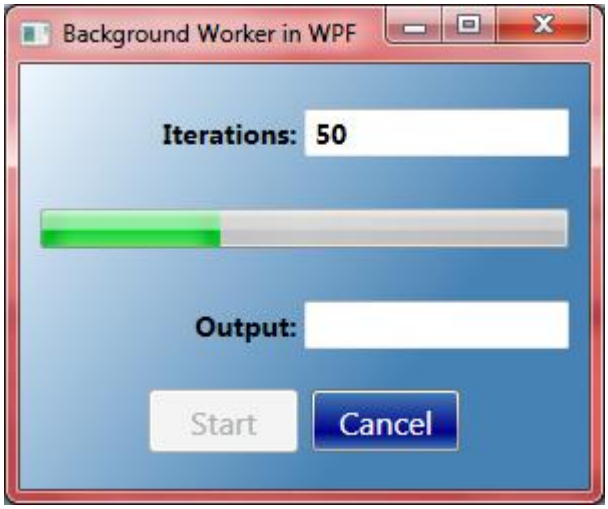
```
private void BackgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    var bgw = sender as BackgroundWorker;
    e.Result = DoSlowProcess((int)e.Argument, bgw, e);
}
```

We'll just cast the sender to a BackgroundWorker and pass it on through. We'll just pass the DoWorkEventArgs parameter through as well.

Finally, we'll add a line of code to the RunWorkerCompleted event to zero out the progress bar after it has completed:

```
private void BackgroundWorker_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else
    {
        outputBox.Text = e.Result.ToString();
        mainProgressBar.Value = 0;
    }
    startButton.IsEnabled = true;
    cancelButton.IsEnabled = false;
}
```

The reason for this is if you are using Windows Vista or Windows 7, the progress bar continues to animate even after it is at 100%. This makes it difficult to tell that the process is complete. So, we'll just set it back to 0 after it's done.



Now, if we run the application again, we'll see that we have a functional progress bar. But sometimes, we need more progress information than simply a percentage.

Additional Progress Information

In addition to the ProgressPercentage, ProgressChangedEventArgs offers a UserState property. The good news is that this property is of type "object", and we use it to pass whatever progress information we want from the background to the UI.

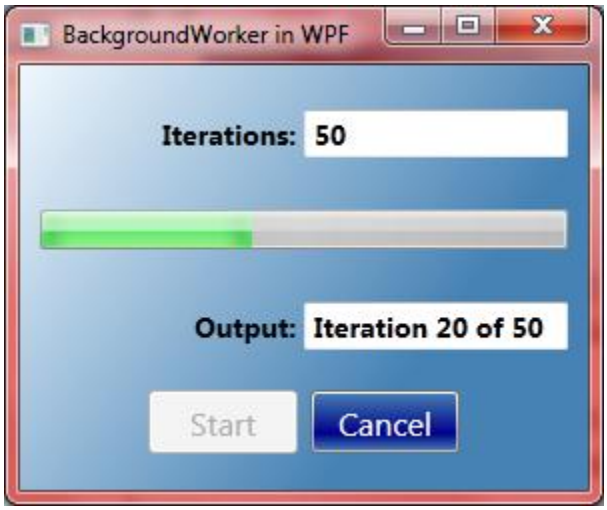
First, let's create an additional message to pass in our progress event. In this case, we'll create a string such as "Iteration 7 of 50". Then we pass this as an additional parameter to the ReportProgress method. Here's the relevant snippet from our DoSlowProcess method:

```
if (worker.WorkerReportsProgress)
{
    int percentComplete =
        (int)((float)i / (float)iterations * 100);
    string updateMessage =
        string.Format("Iteration {0} of {1}", i, iterations);
    worker.ReportProgress(percentComplete, updateMessage);
}
```

Then on the UI side, we can put this message into our output box by using the UserState property:

```
private void BackgroundWorker_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
{
    mainProgressBar.Value = e.ProgressPercentage;
    outputBox.Text = (string)e.UserState;
}
```

Now, if we run the application, we will see the output box change while the process is running:



You can imagine a number of uses for this functionality. For example, if we were processing a set of files in the background, we could report to the UI which file we were currently processing. And since the UserState property is of type "object", we use it for complex types or even collections of types.

The last step to complete the functionality of our application is to add cancellation. But first, we'll take a look at error handling.

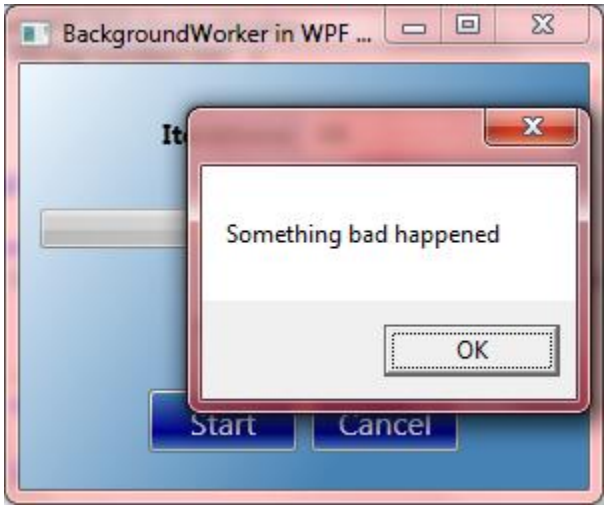
Error Handling

Another feature of the BackgroundWorker component centers around error handling. Exceptions normally stay on their own thread. This means that if you handle threading manually, then any exceptions that happen on the secondary thread will stay on that thread; meaning that your primary thread will not know that an exception occurred unless you build in that error handling yourself.

The BackgroundWorker automatically traps exceptions that happen on the background thread and passes them through to the UI thread. This ends up as the `e.Error` property of the `RunWorkerCompletedEventArgs` that we checked earlier. Let's try this out by adding and exception to the method on our background thread:

```
private int DoSlowProcess(int iterations,
    BackgroundWorker worker, DoWorkEventArgs e)
{
    // throw exception for testing
    throw new System.Exception("Something bad happened");

    int result = 0;
    ...
}
```

If we run the application now, here's what we get when we click the "Start" button:



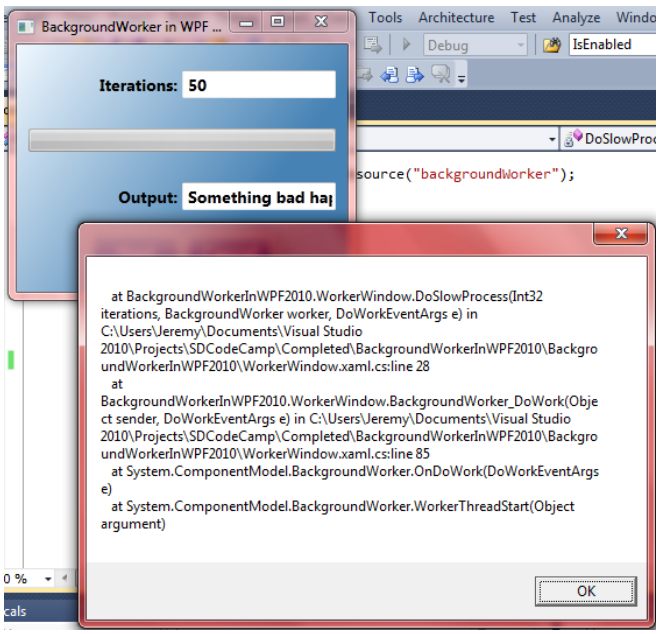
Here's a reminder of the `RunWorkerCompleted` event handler:

```
private void BackgroundWorker_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else
    ...
}
```

You can see that the Message from the Exception we threw in the background thread ("Something bad happened") is the same `e.Error` property here. And just to show that this is the actual Exception coming through and not just some error message, let's expand what we show in our display:

```
private void BackgroundWorker_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
      outputBox.Text = e.Error.Message;
      MessageBox.Show(e.Error.StackTrace);
    }
    else
    ...
}
```

And the output:



So, you can see that we have the actual exception from the background thread available to us on the UI thread. This also means that during debugging, you can breakpoint on the `e.Error` property and get the full complement of information that you get with an Exception.

Note: If you are following along with the code, you will want to comment out the "`throw new System.Exception`" statement before moving on.

The IsBusy Property

One more property of the BackgroundWorker to consider is the `IsBusy` property. The BackgroundWorker component has a limitation of only being able to run one thing at a time. This means that if you call the `RunWorkerAsync` method before the background thread has completed, you will get an Exception. In our application we do not need to worry about this because we disable the "Start" button while the background thread is running. But if you do not do something like this, then you can check the `IsBusy` property to see if the BackgroundWorker is currently running.

A safer way of coding our application would look like this:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    int iterations = 0;
    if (int.TryParse(inputBox.Text, out iterations))
    {
        if (!backgroundWorker.IsBusy)
            backgroundWorker.RunWorkerAsync(iterations);
        startButton.IsEnabled = false;
        cancelButton.IsEnabled = true;
        outputBox.Text = "";
    }
}
```

If you want to see how this works try the following:

1. Comment out the "`if (!backgroundWorker.IsBusy)`" and "`startButton.IsEnabled = false;`" statements.
2. Run the application.
3. Click the "Start" button.
4. Click the "Start" button again before the process has completed (you can easily tell by checking the progress bar).

This will give you an Exception. Now uncomment the "`if (!backgroundWorker.IsBusy)`" statement and try steps 2 – 4 again. You'll see that you won't get an Exception since the process does not try to start again. Checking the `IsBusy` property is the safer way to code.

Canceling the Background Process

Before implementing cancellation in your application, you will need to take a few things into consideration. First, when you cancel a BackgroundWorker process, there is no event that fires, and the process does not stop immediately. Instead, a cancellation flag gets set on the BackgroundWorker. It is up to your long-running process to check for this flag and to stop running if necessary. In my example above with the long-running SQL query, I could not implement cancellation because there was no "iteration" in my process – it was simply waiting for the query to return.

In our example here, since we are using a loop, we have a perfect place to check for cancellation and stop our process. Here's an overview of the steps we'll take, then we'll look at each in more detail.

First, we need to set a property on the BackgroundWorker ("WorkerReportsCancellation"). Then we'll tell the component we want to cancel in the Cancel Button event handler. Next we'll add the cancellation code to our DoSlowProcess method. And finally, we'll make a few changes to the RunWorkerCompleted event handler to behave differently if the process was canceled.

First, the XAML:

```
<Window.Resources>
    <cm:BackgroundWorker x:Key="backgroundWorker"
        DoWork="BackgroundWorker_DoWork"
        RunWorkerCompleted="BackgroundWorker_RunWorkerCompleted"
        WorkerReportsProgress="True"
        ProgressChanged="BackgroundWorker_ProgressChanged"
        WorkerSupportsCancellation="True"/>
</Window.Resources>
```

Next, the Cancel Button event handler:

```
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            backgroundWorker.CancelAsync();
        }
```

You can see that we're simply calling the CancelAsync method of the BackgroundWorker.

Next, add the cancellation logic to the DoSlowProcess:

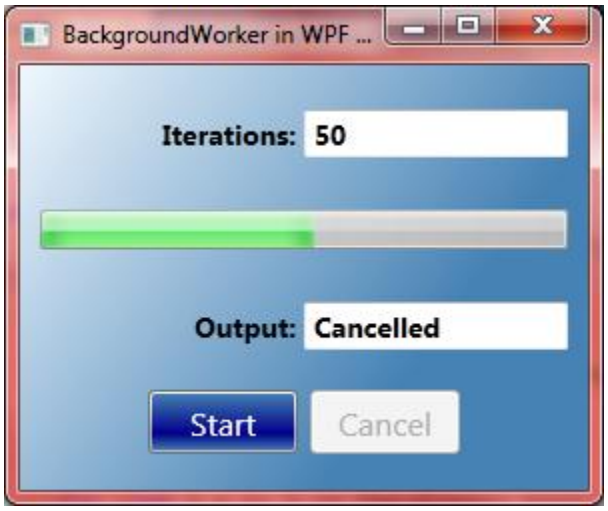
```
private int DoSlowProcess(int iterations,
    BackgroundWorker worker, DoWorkEventArgs e)
{
    int result = 0;

    for (int i = 0; i <= iterations; i++)
    {
        if (worker != null)
        {
            if (worker.CancellationPending)
            {
                e.Cancel = true;
                break;
            }
            if (worker.WorkerReportsProgress)
            {
                int percentComplete =
                    (int)((float)i / (float)iterations * 100);
                worker.ReportProgress(percentComplete);
            }
        }

        Thread.Sleep(100);
        result = i;
    }

    return result;
}
```

You can see that we added another conditional to check CancellationPending. If so, then we'll set the e.Cancel property of the DoWorkEventArgs and return from our method.

And finally, the RunWorkerCompleted:

```
private void BackgroundWorker_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.Message);
    }
    else if (e.Cancelled)
    {
        outputBox.Text = "Cancelled";
    }
    else
    {
        outputBox.Text = e.Result.ToString();
        workerProgress.Value = 0;
    }
    startButton.IsEnabled = true;
    cancelButton.IsEnabled = false;
}
```

Here, you can see that we are checking the e.Cancelled property of the EventArgs. If it's true, then we'll put "Cancelled" in the output box. One thing you'll note: we are not resetting the progress bar in event of cancellation. This is so that if you stop the process, you can still see how far it got before the cancellation.



Now when we run the application, we'll see that we have a long-running process that runs in the background (keeping the UI responsive), an updating progress bar, and a working Cancel button.

RunWorkerCompleted (Almost) Always Fires

Here's something you should have noticed by now: the RunWorkerCompleted event fires regardless of how the background thread completed – whether it completed normally (success), whether it generated an exception (failure), or whether it is cancelled (cancel). You will want to pay careful attention to what you place in this event handler as well as ensuring that you are checking the EventArgs for the completion state.

Now, I did say "almost" always fires. If you shut down your application while the BackgroundWorker is still running, the application will exit immediately without waiting for the background thread to complete.

Wrap Up

The BackgroundWorker component allows us to put long-running processes onto a background thread without the usual complexities of threading. We have seen how we can get progress updates that we can show in our UI as well as how to cancel a process before it has completed. In addition, we've seen that even when updating our UI, we don't have to worry about communicating across threads. It is all handled for us in the component.

Probably the best thing about the BackgroundWorker is that it allows us to get our feet wet in the world of threading in an easy and relatively safe way. Think about this the next time you come across an application that is "Not Responding". And do what you can to keep your UIs responsive for your users.

Happy coding!