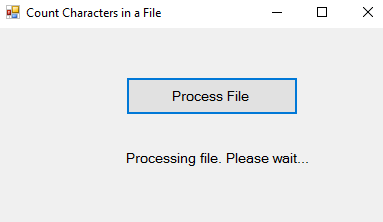
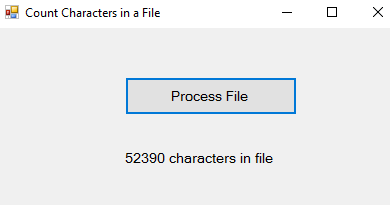
**Async and await in c# example**

Let us create a simple Windows Forms Application that counts the number of characters in a given file. Let us assume the file is very big and it takes around 5 seconds to read and count the number of characters in the file. When the **"Process File Button"** is clicked, the application should display the message **"Processing File. Please wait"**.   
  
  
  
  
  
  
  
As soon as the application finishes processing the file it should display the the number of characters as shown below.  
  
  
  
Another improtant requirement is that the application should remain responsive throughout the entire process, i.e when the application is busy processing the file the application should not hang and we should still be able to interact with the application. We should be able to click with in the other controls on the form, move the form around on the screen, resize it if required etc.  
  
Let us first create the Windows Forms Application without using async and await keywords and see how it behaves. Here are the steps.  
  
1. In your C: drive, create a new folder. Name it **Data**. In the folder create a new Text Document. Name it **Data.txt**. Type some text in the file and save it. The application that we are going to create, counts the number of characters in this file.  
  
2. Create a New **"Windows Forms Application"**. Name it AsyncExample.  
  
3. Drag and Drop a **"Button"** on the Form and set the following properties  
   Name = btnProcessFIle  
   Font - Size = 10  
   Text = Process File  
  
4. Drag and Drop a **"Label"** on the Form and set the following properties  
   Name = lblCount  
   Font - Size = 10  
   Text = ""  
  
5. Double Click on the **"Button"** control to generate the "Click" event handler  
  
6. Copy and paste the following code in Form1.cs

using System;

using System.IO;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace AsyncExample

{

    public partial class Form1 : Form

    {

        public Form1()

        {

            InitializeComponent();

        }

        private int CountCharacters()

        {

            int count = 0;

            // Create a StreamReader and point it to the file to read

            using (StreamReader reader = new StreamReader("C:\\Data\\Data.txt"))

            {

                string content = reader.ReadToEnd();

                count = content.Length;

                // Make the program look busy for 5 seconds

                Thread.Sleep(5000);

            }

            return count;

        }

        private void btnProcessFIle\_Click(object sender, EventArgs e)

        {

            lblCount.Text = "Processing file. Please wait...";

            int count = CountCharacters();

            lblCount.Text = count.ToString() + " characters in file";

        }

    }

}

7. Run the application and click the **"Process File"** button. You will notice the following problems.

* The application does not display the status message i.e "Processing file. Please wait."
* While the application is busy processing the file, it becomes unresponsive. You cannot move the form window or resize it.

These problems can be very easily fixed by using the **async** and **await**keywords. Notice only the **btnProcessFIle\_Click**() event handler method needs to change.

// Make the method async by using the async keyword

private async void btnProcessFIle\_Click(object sender, EventArgs e)

{

    // Create a task to execute CountCharacters() function

    // CountCharacters() function returns int, so we created Task<int>

    Task<int> task = new Task<int>(CountCharacters);

    task.Start();

    lblCount.Text = "Processing file. Please wait...";

    // Wait until the long running task completes

    int count = await task;

    lblCount.Text = count.ToString() + " characters in file";

}

Now, when we click the **"Process File"** button, notice

* The application displays the status message (**"Processing file. Please wait"**) immediately.
* Even when the application is busy processing the file, it is responsive. You can move the form window around or resize it.

**So what is the use of async and await keywords in C#**  
async and await keywords are used to create asynchronous methods. The async keyword specifies that a method is an asynchronous method and the await keyword specifies a suspension point. The await operator signalls that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.  
  
An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn’t cause a compiler error.  
  
**You may have a few questions at this point.**  
1. Can't we achieve the same thing using a Thread.   
2. What is the difference between a Thread and a Task  
3. When to use a Task over Thread and vice-versa

### c# wait for thread to finish without blocking

we discussed creating a simple responsive windows forms application using Task, and async & await keywords. Now we will discuss how to do the same using a Thread instead of Task.

To use a Thread instead of a Task we only need to change btnProcessFile\_Click() method as shown below.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    lblCount.Text = count.ToString() + " characters in file";

}

At this point the application does not work as expected. We have two problems with the above code.  
1. We do not see the message, "Processing file. Please wait." at all  
2. It displays "0 characters in file"  
  
**Why is this happening**  
The Main thread i.e the UI thread has created a worker thread which executes CountCharacters() function. The worker thread takes at least 5 seconds to complete. In the mean time the Main thread continues executing the following 2 lines of code.

lblCount.Text = "Processing file. Please wait...";

lblCount.Text = count.ToString() + " characters in file";

**But why didn't we see the message "Processing file. Please wait..."**  
This is because, the UI thread executes the above 2 lines of code so fast that the second message overwrites the first message and at that speed it is impossible for a human eye to spot the overwriting.  
  
**How to solve the above two problems**  
It is very simple. The Main thread has to wait for the worker thread to finish it's work before the UI thread can display the second message. We achieve this by using Join() method on the worker thread.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    // Join() blocks the Main thread (UI Thread)

    thread.Join();

    lblCount.Text = count.ToString() + " characters in file";

}

At this point run the application and test it. We have fixed the above two problems but introduced a new problem. While the application is busy processning the file, the UI is blocked i.e we cannot move the form around or resize it.  
  
You may be thinking why can't we move the code that updates the label control Text property into the worker thread as shown below. This is dangerous because, the thread that has created the control must modify the control. In our case the Main thread (i.e UI Thread) is the thread that has created the label control so only the Main thread should set it's Text property and not the worker thread. If you run the application it may or may not work as expected. If it is working, it is only working by blind luck. 

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        // This is dangerous

        lblCount.Text = count.ToString() + " characters in file";

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

The right way to achieve this is by using BeginInvoke() method as shown below. BeginInvoke() method asks the UI thread to set the Text property of the label control in a type safe manner.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        Action action = () => lblCount.Text = count.ToString() + " characters in file";

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

In the example above, notice that the Action delegate points to a piece of code. The Action delegate is then passed to the BeginInvoke() method which asks the UI thread to execute that piece of code asynchronously in a type safe manner. The above code can also be rewritten as shown below.

int characterCount = 0;

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    Thread thread = new Thread(() =>

    {

        characterCount = CountCharacters();

        // Action delegate points to SetLabelTextProperty method

        // Signature of SetLabelTextProperty() method should match

        // with the signature of Action delegate

        Action action = new Action(SetLabelTextProperty);

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

private void SetLabelTextProperty()

{

    lblCount.Text = characterCount.ToString() + " characters in file";

}

Asynchronous implementation is very easy with tasks, and async & await keywords. Though the above example is a very simple example, notice the code is already getting relatively complicated. Imagine if we have multiple threads, and we want to use the result of one thread from another thread and so on and so forth. It can get painful and complicated. In our previous video, we have seen how easy it is to achieve exactly the same thing using a Task.