



Lambdas

.NET

**DE HOGESCHOOL
MET HET NETWERK**

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook



Contents

- Delegates / Events
 - Generic version
 - Delegate inference
 - anonymous methods
- Lambdas
 - What?
 - Action<T>
 - Func<T,TResult>
- Real World Examples



This module focuses on implementing delegates and events in a CONCISE (= clear and short) manner. This is accomplished by lambdas, which are introduced by means of anonymous methods.

Last, we conclude with some real world examples where events and delegates are used. Before we go into lambdas, we introduce generic versions of common event types (EventHandler<T>).

Events – Delegates Recap

- A delegate is a type which defines a function pointer
- A delegate has an invocation list
- An event is a special kind of delegate
 - Protected access to the invocation list
 - Convention: object sender and EventArgs args
- Publisher / Subscriber design pattern



We use the example from the Pluralsight Course:

<http://www.pluralsight.com/courses/csharp-events-delegates>

Example (Recap)

```
public delegate void WorkPerformedHandler(object sender, WorkPerformedEventArgs args);

public class Worker
{
    public event WorkPerformedHandler WorkPerformed;
    public event EventHandler WorkCompleted;

    public void DoWork(int hours, WorkType workType)
    {
        for (int i = 0; i < hours; i++)
        {
            System.Threading.Thread.Sleep(1000);
            OnWorkPerformed(i + 1, workType);
        }
        OnWorkCompleted();
    }
}
```



This slides uses the code from: **WorkerWithInference**

WorkPerformedHandler: delegate definition using conventions used by the .NET framework, e.g. sender and args

Two events:

- WorkPerformed
- WorkCompleted → has no information to carry from sender to receiver, so uses default EventArgs class

DoWork:

- Simulates a delay via Sleep → do not use in production code!

Example (Recap)

```
protected virtual void OnWorkPerformed(int hours, WorkType workType)
{
    if (WorkPerformed != null)
    {
        WorkPerformed(this, new WorkPerformedEventArgs(hours, workType));
    }
}

protected virtual void OnWorkCompleted()
{
    if (WorkCompleted != null)
    {
        WorkCompleted(this, EventArgs.Empty);
    }
}
```



OnWorkPerformed / OnWorkCompleted

- Invoke event handler by raising the event and passing the arguments
- EventArgs.Empty : no data
[https://msdn.microsoft.com/en-us/library/system.eventargs.empty\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.eventargs.empty(v=vs.110).aspx)
- null check, because delegates could be null
This could be rewritten using the ? (Null-conditional operator)
More info: <http://www.kunal-chowdhury.com/2014/12/csharp-6-null-conditional-operators.html#E0kMDxagcchjgg7y.97>
e.g.:

WorkCompleted?.(this, EventArgs.Empty)

- protected virtual : these methods could be overridden, this is not an important element for the remainder of the discussion.

Delegate Inference

```
worker.WorkPerformed += new WorkPerformedHandler(Worker_WorkPerformed);
```



```
worker.WorkPerformed += Worker_WorkPerformed; // inference
```



Demo: WorkerWithInference

Instead of writing the full delegate constructor, the compiler can **derive** the correct signature of the handler and therefore you only have to specify the method to be added to the invocation list.

Using Generics

```
public delegate void WorkPerformedEventHandler(object sender, WorkPerformedEventArgs args);
```

```
public class WorkPerformedEventArgs : EventArgs {  
    public WorkPerformedEventHandler Handler { get; set; }  
}
```

```
public event EventHandler<WorkPerformedEventArgs> WorkPerformed;
```




If you think about it, using events always involves writing a custom delegate, but the only thing that differs is the derived EventArgs class (e.g. WorkPerformedEventArgs).

Therefore you could use the generic class **EventHandler<T>** where you specify the derived EventArgs class between the < and >

[https://msdn.microsoft.com/en-us/library/db0etb8x\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/db0etb8x(v=vs.110).aspx)

Attach Directly

```
var worker = new Worker();  
worker.WorkPerformed += Worker_WorkPerformed;
```



```
static void Worker_WorkPerformed(object sender, WorkPerformedEventArgs args)  
{  
    Console.WriteLine("Work performed: " + args.Hours);  
}
```



If the code for the event handler is used only once (as is often the case), it makes sense to attach the code immediately without creating a (named) method first.

You can do this using an anonymous method. This is not used so often anymore.

Anonymous methods

```
var worker = new Worker();  
worker.WorkPerformed += delegate(object sender, WorkPerformedEventArgs e)  
{  
    Console.WriteLine("Work performed: " + e.Hours);  
};
```



Anonymous methods allow event handler code to be hooked directly to an event. They are defined using the **delegate** keyword.

Advantage: for short methods it makes the code more readable.

Disadvantage: the method implementation is not reusable.

Demo: WorkerAnonymous

XAML vs Anonymous vs Lambda

```
<Grid HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button x:Name="SubmitButton1" Click="SubmitButton_Click" Content="Button 1"/>
</Grid>
```

```
private void SubmitButton_Click(object sender, RoutedEventArgs e)
{
    var b = sender as Button;
    MessageBox.Show(String.Format("You clicked {0}", b.Name));
}
```



Demo: SubmitButtonClick

XAML

The Click attribute corresponds with a Click Event which defines a delegate from type `RoutedEventHandler`:

[https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.buttonbase.click\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.buttonbase.click(v=vs.110).aspx)

<https://msdn.microsoft.com/en-us/library/cc545009.aspx>

The XAML syntax adds your method to the invocation list of this event.

XAML vs Anonymous vs Lambda

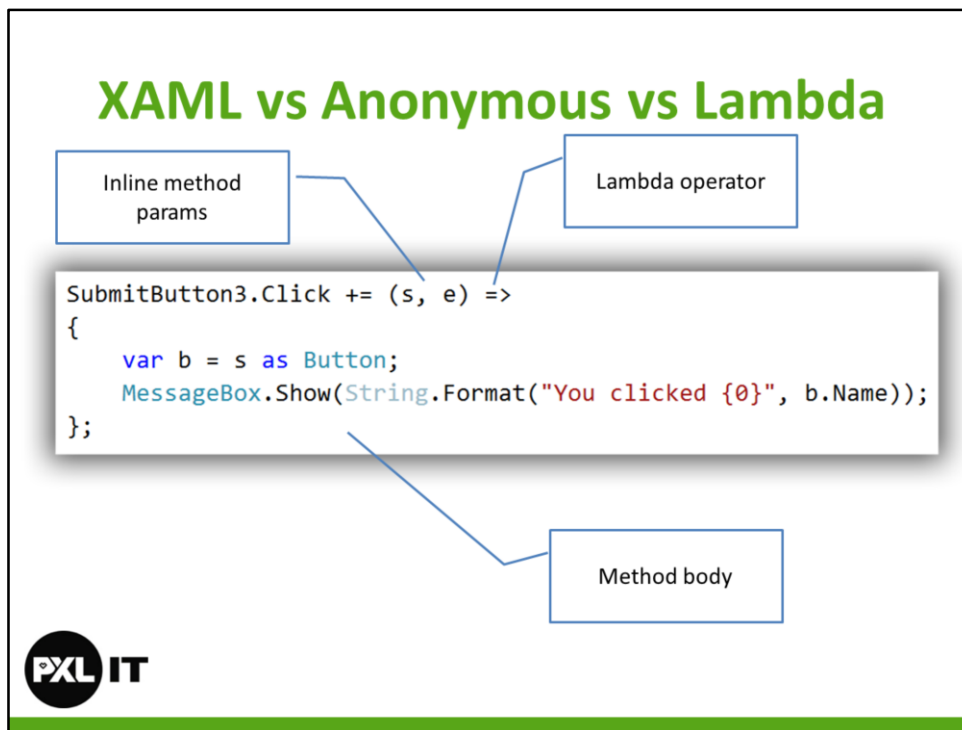
```
SubmitButton2.Click += delegate (object sender, RoutedEventArgs e)
{
    var b = sender as Button;
    MessageBox.Show(String.Format("You clicked {0}", b.Name));
};
```



Demo: SubmitButtonClick

C# Anonymous methods

You could also write the event handler using anonymous methods. This is not used so often anymore since the introduction of lambdas in 2007 (!)



Demo: SubmitButtonClick

Or you use a lambda, this gives the most elegant (concise) code.

A lambda has 3 parts:

(s, e) are the **inline method parameters**. There is **no** type declaration, because the compiler derives these based on the event (and underlying delegate) type.

=> is the **lambda operator**. It separates the method parameters from the body.

Method body is the code block that will be executed eventually and can make use of the parameters that are passed in. Note: { } and ;

Assigning a Lambda to a Delegate

```
delegate int AddDelegate(int a, int b);

static void Main(string[] args)
{
    AddDelegate ad = (a, b) => a + b;
    int result = ad(1, 1); // result = 2
}
```



Demo: LambdaDelegate

If you have a delegate, you can add a method to it by means of a lambda expression.

- Compiler detects the parameter types
- Names a, b don't have to be the same, e.g: AddDelegate ad = (foo1, foo2) => foo1 + foo2;
- No return keyword necessary, because there is no { } → just one statement → this is supposed to return a value

Handling empty parameters

```
delegate bool LogDelegate();

static void Main(string[] args)
{
    LogDelegate ld = () =>
    {
        UpdateDatabase();
        WriteToEventLog();
        return true;
    };

    bool status = ld();
}
```



Demo: LambdaDelegate

Demo: Lambdas with events

```
var worker = new Worker();
worker.WorkPerformed += (s, e) =>
{
    Console.WriteLine("Work performed: " + e.Hours);
};

worker.WorkCompleted += (s, e) =>
{
    Console.WriteLine("Worker is done");
};
```



Demo: WorkerLambdas

This is almost exact the same code as using an anonymous method, but thanks to lambdas, parameter declaration is much shorter:

`(s, e) =>` *instead of* `delegate(object sender, WorkPerformedEventArgs e)`

Delegates in .NET

- The .NET framework provides several different delegates that provide flexible options:
 - **Action<T>** - Accepts a single parameter and returns no value
 - **Func<T,TResult>** - Accepts a single parameter and returns a value of type TResult



Built-in delegates that save you code.

Generics because types can differ.

Using Action<T>

```
C# C+ static void Main(string[] args)
{
    public Action<string> messageTarget;

    if (args.Length > 1)
    {
        messageTarget = ShowWindowsMessage;
    }
    else
    {
        messageTarget = Console.WriteLine;
        //messageTarget = new Action<string>(Console.WriteLine);
    }

    messageTarget("Invoking Action!");
}

private static void ShowWindowsMessage(string message)
{
    MessageBox.Show(message);
}
```



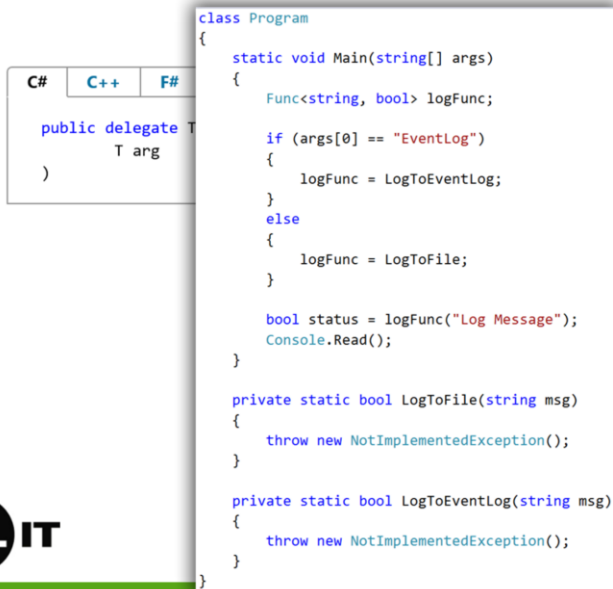
Demo: ActionMessages

[https://msdn.microsoft.com/en-us/library/018hxda8\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/018hxda8(v=vs.110).aspx)

Action<T> encapsulates a method that has a single parameter and does not return a value.

You pass in a method like a “self-written” delegate. By using generics, you can pass in different types of parameters.

Using Func<T, TResult>



The screenshot shows a code editor with a language selector at the top (C#, C++, F#) and a code file named 'Program.cs'. The code defines a static class 'Program' with a 'Main' method. The 'Main' method uses a 'Func<string, bool>' delegate to call either 'LogToFile' or 'LogToEventLog' based on the command-line argument. Both logging methods are currently implemented to throw a 'NotSupportedException'.

```
class Program
{
    static void Main(string[] args)
    {
        Func<string, bool> logFunc;

        if (args[0] == "EventLog")
        {
            logFunc = LogToEventLog;
        }
        else
        {
            logFunc = LogToFile;
        }

        bool status = logFunc("Log Message");
        Console.Read();
    }

    private static bool LogToFile(string msg)
    {
        throw new NotImplementedException();
    }

    private static bool LogToEventLog(string msg)
    {
        throw new NotImplementedException();
    }
}
```



Demo: FuncMessages

[https://msdn.microsoft.com/en-us/library/bb549151\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb549151(v=vs.110).aspx)

Func<T, TResult> encapsulates a method that has one parameter and returns a value of the type specified by the *TResult* parameter.

You pass in a function like a “self-written” delegate. By using generics, you can pass in different types of parameters and result types.

Demo: WorkerActionFunc

```
public delegate int BizRulesDelegate(int x, int y);

class Program
{
    static void Main(string[] args)
    {
        // using custom delegates
        var data = new ProcessData();
        BizRulesDelegate addDel = (x, y) => x + y;
        BizRulesDelegate multiplyDel = (x, y) => x * y;
        data.Process(2, 3, multiplyDel);

        // using Actions
        Action<int, int> myAction = (x, y) => Console.WriteLine(x + y);
        Action<int, int> myMultiplyAction = (x, y) => Console.WriteLine(x * y);
        data.ProcessAction(2, 3, myAction);
        data.ProcessAction(2, 3, myMultiplyAction);

        // using Func
        Func<int, int, int> funcAddDel = (x, y) => x + y;
        Func<int, int, int> funcMultiplyDel = (x, y) => x * y;
        data.ProcessFunc(2, 3, funcAddDel);
        data.ProcessFunc(2, 3, funcMultiplyDel);
    }
}
```



Demo: WorkerActionFunc

Step through the code and make sure you understand every line of it.

Lambdas and LINQ

- LINQ uses extension methods on object collections
- These extension methods take delegates as arguments to filter data
- These delegates are expressed most elegantly as lambdas
- Demo: LambdasLINQ



Extension methods

- You want to add some members to a type
- You don't need to add extra data to the type
- You can't change the type itself
 - You don't have the code
 - You don't want to break existing code
- They are in fact a special kind of static methods



More info: <https://msdn.microsoft.com/en-us/library/bb383977.aspx>

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

Extension Methods

```
// Define an extension method in a non-nested static class.  
public static class Extensions  
{  
    public static Grades minPassing = Grades.D;  
    public static bool Passing(this Grades grade)  
    {  
        return grade >= minPassing;  
    }  
}
```



The class itself is static, and the extension method also.

By using the **this** keyword, you “extend” the method towards the object instance, making it appear like a normal method the struct Grade.

Extension Methods

```
public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
class Program
{
    static void Main(string[] args)
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

        Extensions.minPassing = Grades.C;
        Console.WriteLine("\r\nRaising the bar!\r\n");
        Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
        Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
    }
}
```



This example can be found in the MSDN docs:

<https://msdn.microsoft.com/en-us/library/bb383974.aspx>

Demo: LambdasLINQ

```
var custs = new List<Customer>
{
    new Customer { City = "Phoenix", FirstName = "Jo"},
    new Customer { City = "Phoenix", FirstName = "Ja"},
    new Customer { City = "Seattle", FirstName = "Su"},
    new Customer { City = "New York City", FirstName = "Mi"},
};

var phxCusts = custs
    .Where(c => c.City == "Phoenix" && c.ID < 500)
    .OrderBy(c => c.FirstName);

foreach (var cust in phxCusts)
{
    Console.WriteLine(cust.FirstName);
}
```



Definition of the **Where** extension method:

[https://msdn.microsoft.com/en-us/library/bb534803\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb534803(v=vs.110).aspx)

Definition of the **OrderBy** Extension method:

[https://msdn.microsoft.com/en-us/library/bb534966\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb534966(v=vs.110).aspx)

Arguments of these extension methods are passed in as lambda expressions.

Some Cases

EVENTS AND DELEGATES IN ACTION



Study the demos in Pluralsight yourself.

Communicating between components

Job Management

Job

Edge Park

Employees on Job: Edge Park

1 John Doe

ID2

TitleEdge Park

Start Date9/29/2015

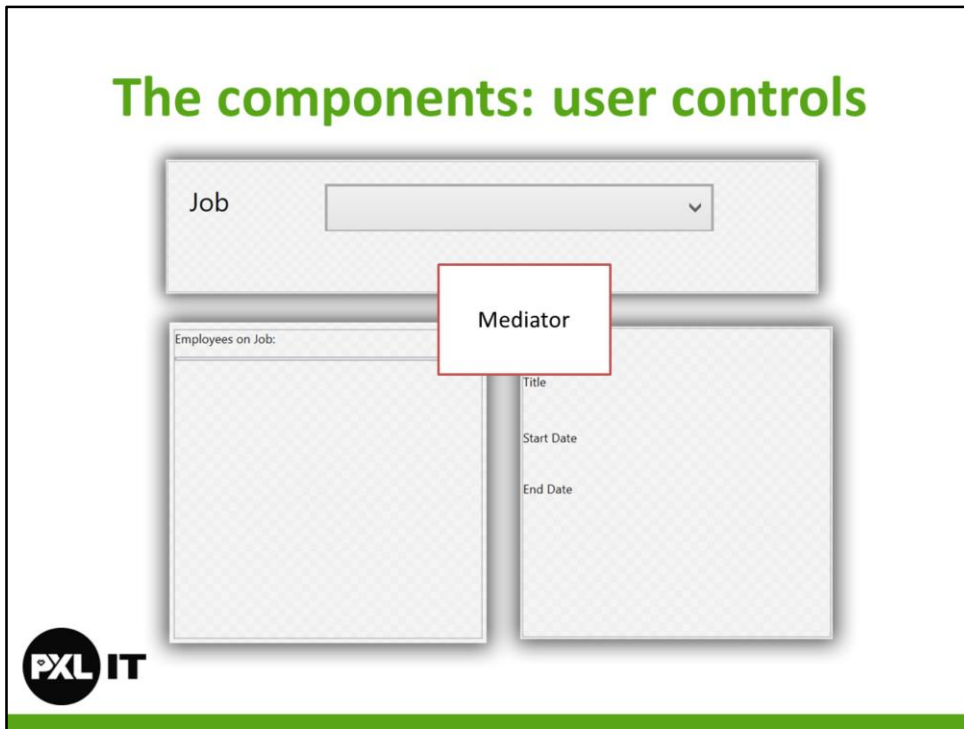
End Date10/9/2015



The problem:

Different areas of the screen about the same data should work together. If you select a different job, the list of Employees should update (left) AND the job details (RIGHT).

The components: user controls



A User Control is simply a collection of (WPF) elements grouped together, so it can be reused in different places.

We want to design this in a loosely coupled way: the “Employees on Job” control and “Job Details” control may not know about the Job combobox. How could this be done?

One possible way to realize this is using the **mediator** design pattern.

Mediator: <http://www.dofactory.com/net/mediator-design-pattern>

They all know about the mediator object, but not about each other

Demo: CommunicatingBetweenControls

The Mediator creates events of type `JobChanged`, which the User Controls subscribe to. The combobox actually changes the Job, so is considered the sender of the event.

NOTE: This is not the only way to solve this problem. Another popular pattern is the Model-View-ViewModel pattern.

Async delegates: bad

- You call a delegate using “BeginInvoke”
- This starts a new thread
- In this thread you NEVER may update gui components directly



More info on **BeginInvoke**:

[https://msdn.microsoft.com/en-us/library/2e08f6yc\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/2e08f6yc(v=vs.110).aspx)

Demo: AsyncBadWPF

```
delegate void UpdateProgressDelegate(int val);

public MainWindow()
{
    InitializeComponent();
}

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    var progDel = new UpdateProgressDelegate(StartProcess);
    // call this asynchronous: spin in another thread
    progDel.BeginInvoke(100, null, null);
    MessageBox.Show("Done with operation!");
}

private void StartProcess(int max)
{
    this.pbStatus.Maximum = max;
    for (int i = 0; i < max; i++)
    {
        Thread.Sleep(10);
        lblOutput.Text = i.ToString();
        this.pbStatus.Value = i;
    }
}
```



Demo: AsyncBadWPF

You call BeginInvoke on the delegate → starts another thread → execute StartProcess

StartProcess → change Text property of lblOutput and Value property of pbStatus → NOT ALLOWED!

Exception: System.InvalidOperationException

Demo: AsyncGoodWPF

- Dispatcher: provides services for managing a queue on a Thread => associated with GUI
- Dispatcher.CheckAccess: is the currently running Thread the GUI thread?
- Dispatcher.BeginInvoke: invoke the delegate on the Thread associated with the Dispatcher => the GUI thread
- Solution: introduce an extra delegate



CheckAccess:

[https://msdn.microsoft.com/en-us/library/ms591588\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms591588(v=vs.110).aspx)

BeginInvoke:

[https://msdn.microsoft.com/en-us/library/cc190824\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/cc190824(v=vs.110).aspx)

Demo: AsyncGoodWPF

```
private delegate void ShowProgressDelegate(int val);
private delegate void StartProcessDelegate(int val);

public MainWindow()
{
    InitializeComponent();
}

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    //Call long running process
    StartProcessDelegate startDel = new StartProcessDelegate(StartProcess);
    //startDel.BeginInvoke executes delegate on new thread
    startDel.BeginInvoke(100, null, null);

    //Show message box to demonstrate that StartProcess()
    //is running asynchronously
    MessageBox.Show("Called after async process started.");
}

// Called Asynchronously
private void StartProcess(int max)
{
    ShowProgress(0);
    for (int i = 0; i <= max; i++)
    {
        Thread.Sleep(10);
        ShowProgress(i);
    }
}
```



Demo: AsyncGoodWPF

Demo: AsyncGoodWPF

```
private void ShowProgress(int i)
{
    //On helper thread so invoke on UI thread to avoid
    //updating UI controls from alternate thread
    if (!Dispatcher.CheckAccess())
    {
        ShowProgressDelegate del = new ShowProgressDelegate(ShowProgress);
        //this.BeginInvoke executes delegate on thread used by form (UI thread)
        Dispatcher.BeginInvoke(del, new object[] { i });
    }
    else
    { //On UI thread so we are safe to update
        this.lblOutput.Text = i.ToString();
        this.pbStatus.Value = i;
    }
}
```



Set a BreakPoint on the first line of this method.

Show Threads (Debug > Windows > Thread)

Watch how this code gets invoked from different threads.

BackgroundWorker

- Executes an operation on a separate thread
- Delegates:
 - **Do_Work** : execute the time consuming operation on a separate thread. Be careful not to update the GUI
 - **ProgressChanged** : runs on the gui thread, so you can report progress
 - **RunWorkerCompleted**: runs on the gui thread, so you can report completion



More info: [https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker(v=vs.110).aspx)

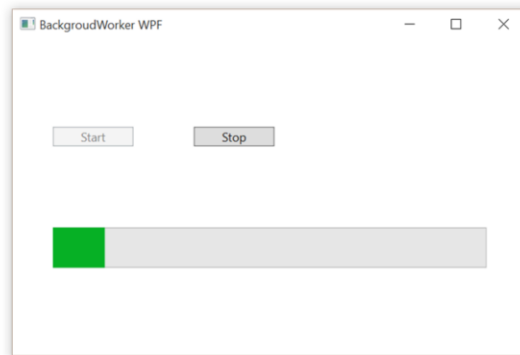
To execute a time-consuming operation in the background, create a BackgroundWorker and listen for events that report the progress of your operation and signal when your operation is finished.

To set up for a background operation, add an event handler for the [DoWork](#) event. Call your time-consuming operation in this event handler. To start the operation, call [RunWorkerAsync](#). To receive notifications of progress updates, handle the [ProgressChanged](#) event. To receive a notification when the operation is completed, handle the [RunWorkerCompleted](#) event.

Note:

You must be careful not to manipulate any user-interface objects in your [DoWork](#) event handler. Instead, communicate to the user interface through the [ProgressChanged](#) and [RunWorkerCompleted](#) events.

BackgroundWorkerDemoWPF



Study the code of the demo yourself.