* 1. 

Hands-On Lab

Module 02: Introduction to Visual Studio / C#

Lab version: 1.0.0

Last updated: 10/14/2011

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS, Windows, MSDN, Visual Basic, Visual C++, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)Microsoft Biology Initiative Training by [Microsoft Corporation](http://research.microsoft.com/bio) is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/). Based on a work at [research.microsoft.com/bio](http://research.microsoft.com/bio).

[Module 02: Introduction to Visual Studio / C# 1](#_Toc293908108)

[Introduction to Visual Studio /C# 4](#_Toc293908109)

[What you should know 5](#_Toc293908110)

[System Requirements 5](#_Toc293908111)

[Task 1: Creating a Console Application 6](#_Toc293908112)

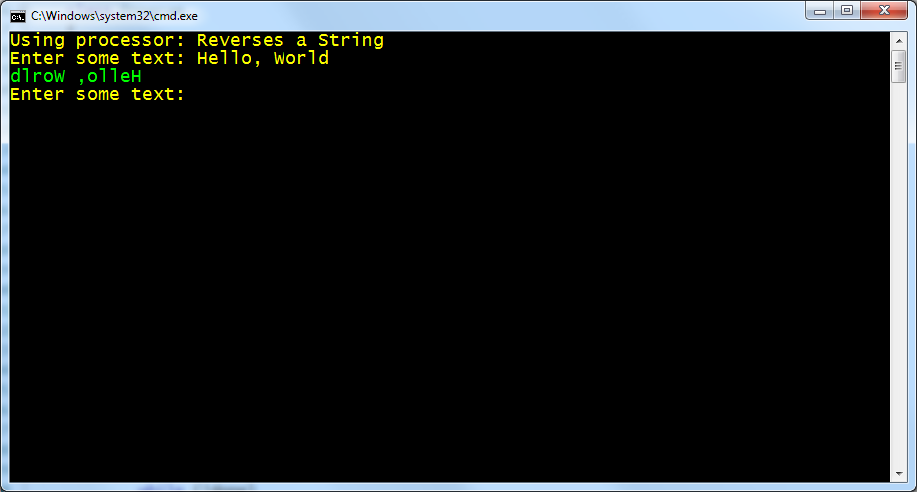
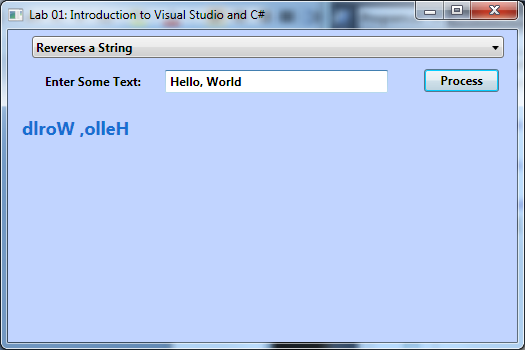
[Task 2: Creating a Class Library 10](#_Toc293908113)

[Lab Solution 18](#_Toc293908114)

[Task 3: Creating a GUI application 19](#_Toc293908115)

[Lab Solution 32](#_Toc293908116)

Introduction to Visual Studio /C#

* 1. This hands-on lab introduces you to the basic features of Visual Studio 2010 – creating solutions and projects, adding libraries and debugging programs. It utilizes C# as the language, although you can experiment with other languages if you choose. It is expected that this exercise will take about an hour to complete fully.
  2. In this exercise you will build two very simple applications that perform actions on input strings. You will build two versions of the application – one that utilizes the command line and console input:
  3. 
  4. and one that uses Windows Presentation Foundation:
  5. 

# What you should know

* 1. The lab assumes very little knowledge about Visual Studio or C# -- if you have some experience using the toolset and language already, then consider just trying to duplicate the above output and below objectives without following the instructions directly – often experiementation leads to discovery!   
       
     Objectives
  2. In this Hands-On Lab, you will get some experience building applications with Visual Studio 2010
  + Create a Console Application and interact with the user through the terminal window.
  + Create interfaces and types that implement those interfaces in C#.
  + Create a class library to hold common (shared) code.
  + Create a simple WPF application using the shared library.

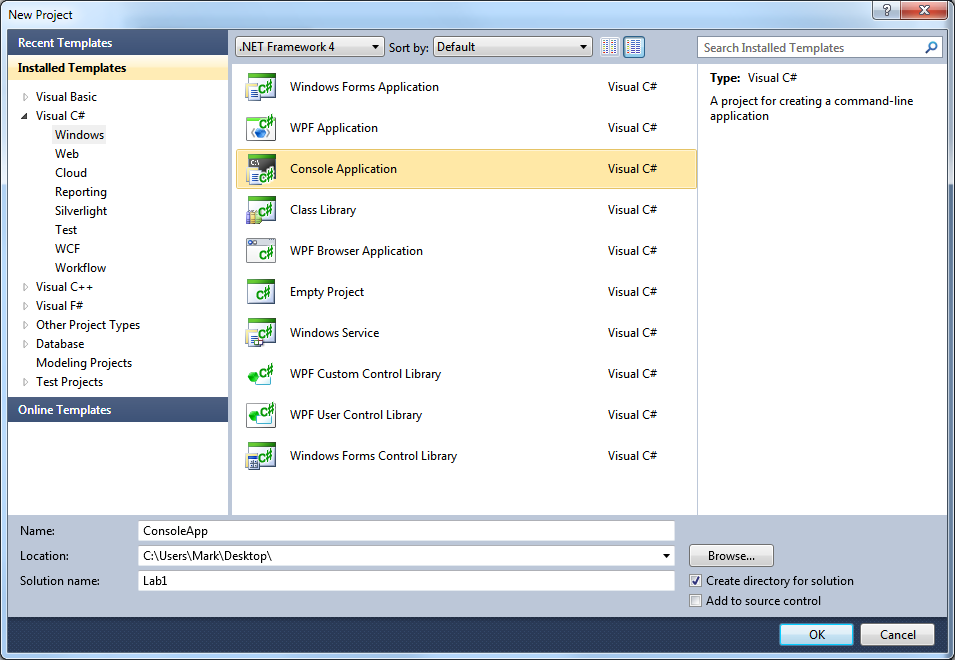
# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + Windows 7 or Windows Vista

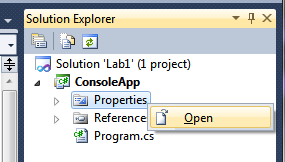
Task 1: Creating a Console Application

* 1. In this first task you will use Visual Studio 2010 to build a Console based application that will interact with the user through a terminal window.

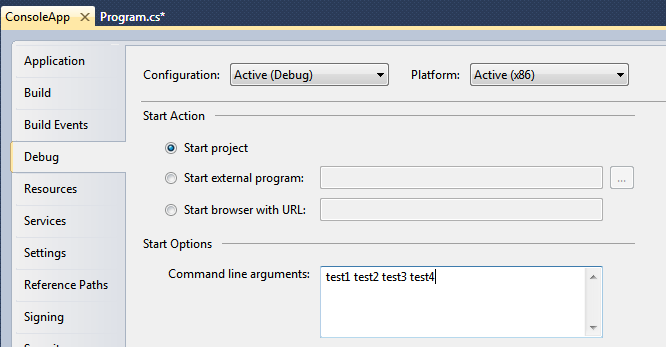
1. To begin the exercise, launch Visual Studio 2010 – it should be in your **Start Menu** 🡪 **All Programs** 🡪 **Microsoft Visual Studio 2010**.
   1. If it prompts you for developer settings (it does this the first time you open the application), make sure to choose “General Development Settings”.
2. Start by creating a new project – **File** 🡪 **New** 🡪 **Project…**
3. In the **New Project** Dialog:
   1. Locate the **Visual C#** section in the **Installed Templates** section.
   2. Make sure .NET Framework 4 is selected in the top combo box.
   3. Select Console Application from the list of templates.
   4. Name the application “ConsoleApp”.
   5. Place it into a directory you have access to (the Desktop is a good choice).
   6. Name the solution “Lab1”.
   7. Check the “**Create Directory for solution**” box – this will create a sub-directory for your project under the solution directory.
   8. Here’s what your screen should look like:



1. Click **OK** to create the project.
2. Visual Studio will create the project and open the single source file – **Program.cs**
   1. Here you will find the main entry point to the console application.
3. As a starting point, try printing all the arguments to the console
   1. Use a **foreach** loop to iterate through the command line arguments array (args).
   2. Use **Console.WriteLine** to print each string to the console on its own line.
   3. When completed, your code should look something like:
   4. class Program
   5. {
   6. static void Main(string[] args)
   7. {
   8. foreach (string argument in args)
   9. {
   10. Console.WriteLine(argument);
   11. }
   12. }
   13. }
4. Now, either open a command prompt (**Start 🡪 All Programs 🡪 Microsoft Visual Studio 2010 🡪 Tools 🡪 Command Prompt**) or you can add a command line to the project properties (see step 8).
   1. Run the program with various command lines to see how it prints out.
5. Or, you can add a command line to the project properties – this will allow you to debug the application and step through it to see how it runs.
   1. Right-click on the **Properties** folder and select “**Open**”, or double-click on the **Properties** folder.



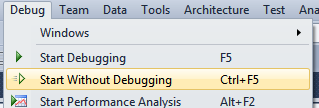
* 1. Switch to the **Debug** tab (on the left of the properties window). Enter some strings in the Command line arguments box:



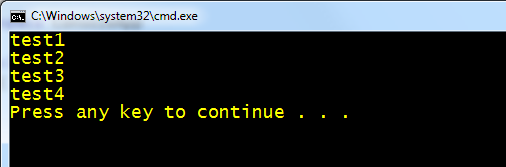
* 1. Look at the tab at the top of the pane – notice it has an asterisk next to the **ConsoleApp** name – this indicates the file it is displaying (properties in this case) has been changed and needs to be saved. It will do that automatically when you close the pane. Close the window by clicking the “X” in the tab:



* 1. You can run the application (without debugging) by pressing CTRL+F5, or using the **Start without Debugging** option in the **Debug** Menu

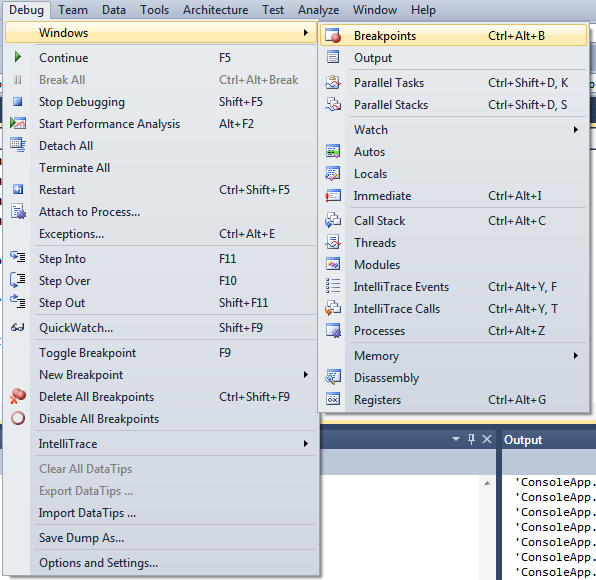


* 1. It should run the application using your typed command line, a nice feature of Visual Studio is it will prompt you to close the window so you can see the output:



* 1. Press a key in the window to close it.

1. Now, try stepping through the application – there are really two primary ways to start debugging the application
   1. Press F10 (or **Debug 🡪 Step Over**) to start the application and stop on the first executable line.
   2. Press F5 (or **Debug 🡪 Start Debugging**) to run the application under the debugger. In this case it will execute until it either hits a breakpoint, unhandled exception, or the program terminates.
2. Try both options – you can use F10 and F11 to step over or into functions, and F9 to place breakpoints to stop the program at a specific location. All of these have equivalent menu commands in the **Debug** menu as well.
   1. When the debugger is stopped, try hovering over a variable with the mouse cursor – Visual Studio will display a debugger tooltip showing you the current value for the variable.
   2. You can also open debugger specific windows – available in the **Debug 🡪 Windows** sub-menu, a few are present when the application is not debugging, a lot more when you are in an active debug session:



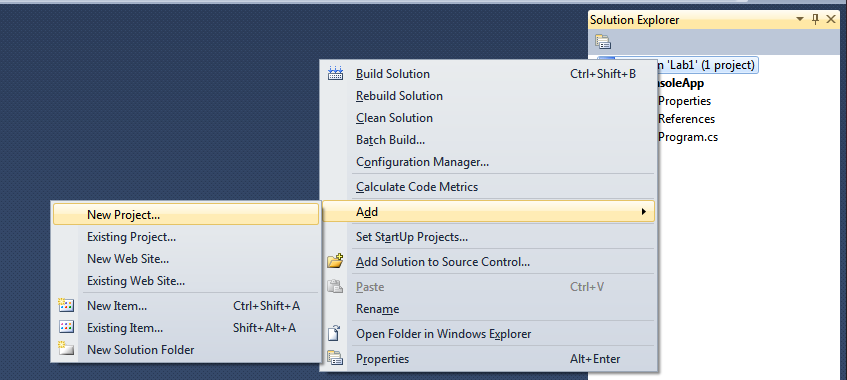
* 1. Feel free to play with these various options to see what information they provide.

There is a final solution to this project available at [Task1.after\Lab1.sln](Task1.after/Lab1.sln) if you need more help.

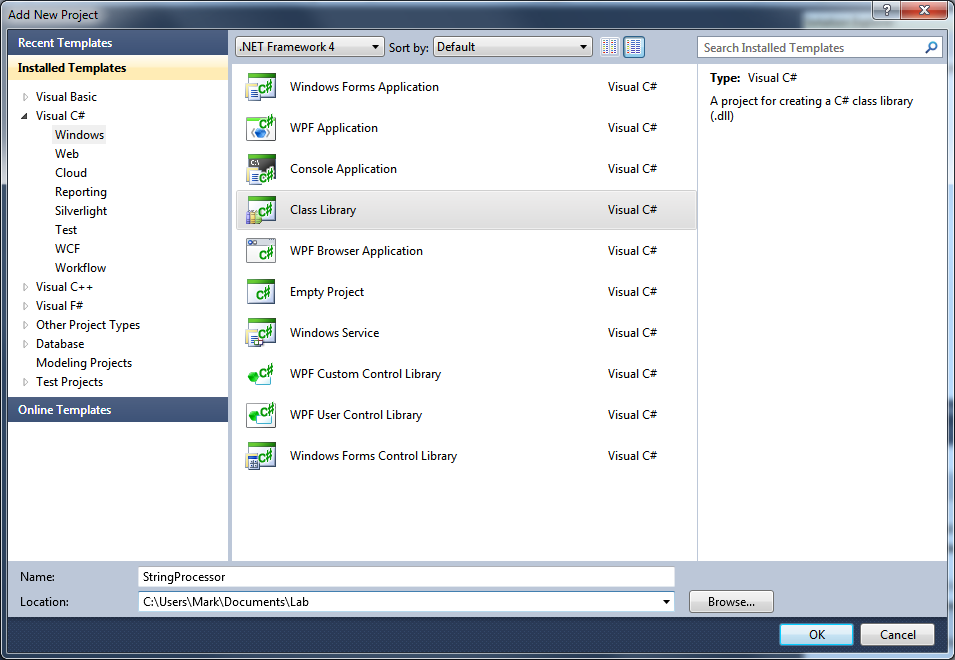
Task 2: Creating a Class Library

In this task you will create another project and add it to your solution. This project will include some interfaces and classes we want to share across projects so we will use a Class Library.

1. To start out, either open the final solution from Task 1 – located at [Task1.after\Lab1.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2001\Lab\Task1.after\Lab1.sln), or continue from task 1. You should have a single Console application project in your solution (named **ConsoleApp**).
2. Create a new project (**File 🡪 New 🡪 Project**, or right-click on the solution and select **Add 🡪 New Project…**)



1. Make sure you have C# highlighted, and select “Class Library” as the project type from the list of templates.
   1. Name the new project “StringProcessor” and place it alongside your current project.



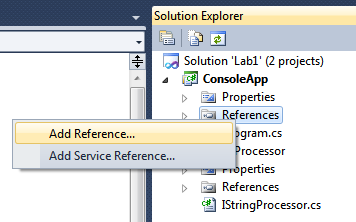
1. Click **OK** to create the project.
2. The project template generates a single file – **Class1.cs** in the library. Rename the file to **IStringProcessor.cs** – either by right-clicking on the file and selecting “**Rename**”, or select the file in the solution explorer and hit “F2” to rename it.
3. Visual Studio should prompt you that you are renaming the file and ask if you want to also rename the class inside it. It is doing this because there is only one class contained in the source file and the name of the class matches the filename. This is generally a convention (one class per source file) people tend to follow and this auto-rename is a nice refactoring feature built into Visual Studio. Go ahead and click “**Yes**” to rename the class too.
4. Now, change the type from a class to an interface.
5. Add a single method to the new interface
   1. Name it “Process”.
   2. Have it take a single string as input and return a string.

Remember an interface denotes a contract – not a specific implementation. The idea is we can have multiple “string processors”, each doing something different to the string – but all of them will have a method called **Process** that takes an input string, does something to it and returns the results.

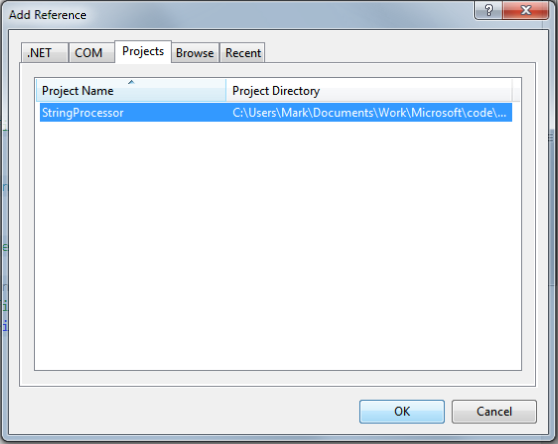
1. Now document your new interface – type three single forward-slash characters together “**///**” over the type or the method – Visual Studio will automatically expand that out to a document header.
2. When you are finished, your interface should look something like this:
   1. /// <summary>
   2. /// This interface defines our string processing
   3. /// contract.
   4. /// </summary>
   5. public interface IStringProcessor
   6. {
   7. /// <summary>
   8. /// Method to process a string.
   9. /// </summary>
   10. /// <param name="input">Input string</param>
   11. /// <returns>Modified (processed) results</returns>
   12. string Process(string input);
   13. }
3. Now close the file and we will create an implementation of the interface in the main project.

Before we can use the string processor interface, we need to tell the compiler that we want to include this new class library as part of the compile process – so it knows to search the generated assembly for types. We do this by adding a reference to the library.

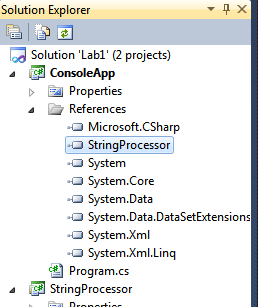
1. Right click on the **References** folder in the **ConsoleApp** project and select “**Add a Reference…**”



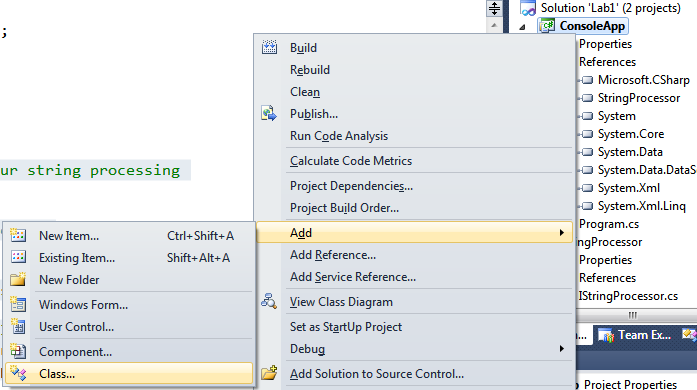
1. Click on the **Projects** tab in the **Add Reference** dialog. You should see the **StringProcessor** project listed there. This tab lists all the projects that are part of this solution. This tab and the .NET tab (which has all the system assemblies in it) are the two places you most commonly go to find new references.
2. Select the **StringProcessor** project and click **OK** to add a reference to the console app.



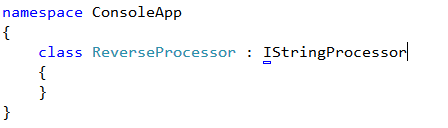
1. You can expand the References folder of the ConsoleApp to verify it is now included as a reference if you like.



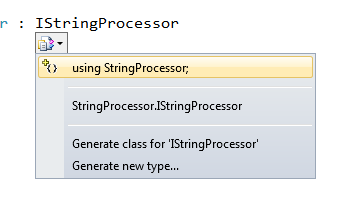
1. Now, we will add a new source file to our console application – right click on the project and select **Add 🡪 Class** (you can also use the **Project 🡪Add Class** menu option, or even **Add New Item** (CTRL+SHIFT+A and select class from the list).



1. Name the class **ReverseProcessor.cs**
2. Add the **IStringProcessor** interface as an implemented interface on the **ReverseProcessor** class, but do not add the actual method yet. Instead, hover your mouse over the **IStringProcessor** string and notice the blue underline under the letter “I”. When you see markers like this it generally means Visual Studio has some functionality you can use here.



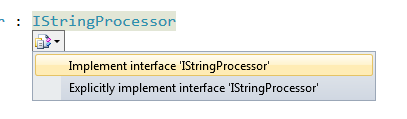
1. Pause your mouse over the blue line (hover on the “I”) and click on the drop down that is displayed. You should see a list of choices – these represent changes Visual Studio will make to your code in order to make it compile properly:



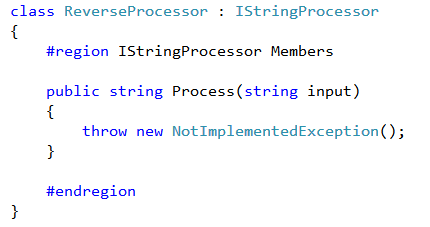
1. The first choice indicates adding the proper **using** declaration to your source file.   
   The second indicates fully supplying the namespace to make the type come into scope.   
   Either works, but the completed lab will choose the first option.
2. Notice the type now changes colors, indicating it is a recognized type, however the underline remains present:



1. Hover on the “I” and click the option again:

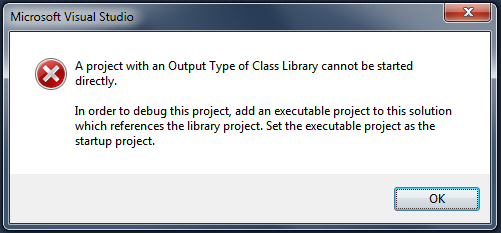


1. Now Visual Studio is offering to actually provide implementations for each of the required methods of **IStringProcessor**! Select the first option – the code should now look like:

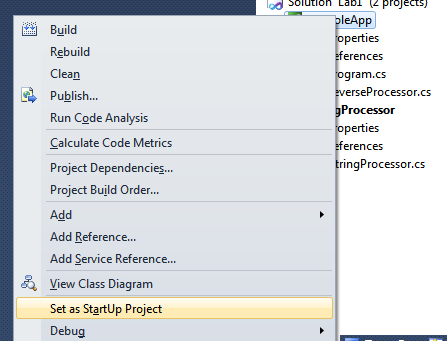


1. It has provided a simple implementation – just throw a **NotImplementedException**. It has also wrapped the method in a #**region** declaration. These are simply editor tags that you can collapse or expand. For very large source files it can be convenient to group related sections together and wrap them in a #**region** tag so you can hide the entire group together. They do not impact the compile process at all.
2. Now remove the default implementation supplied by Visual Studio and replace it with the following:
   1. public string Process(string input)
   2. {
   3. char[] charInput = input.ToCharArray();
   4. Array.Reverse(charInput);
   5. return new string(charInput);
   6. }
3. This code converts the string into a fixed array of characters, reverses that array in place, and then generates a new string from the characters. Remember that strings are actually immutable – they cannot be changed without replacing the string so it is impossible to actually reverse a string in-place. That is why we have taken this roundabout approach.
   1. Eagle eyed developers may have noticed a **Reverse** method on the string itself. It pops up in the Intellisense if you are looking at the methods. This is actually not a method on string, it’s an *extension* method that is added to the string type because it exposes the **IList<char>** interface – that is, it looks like a collection. The **Reverse** method does not exactly do what the implies. The string is still immutable, but the **Reverse** extension method will returns the string’s characters in reverse order as a sequence of characters. You could use this method instead of the first two lines – that is, replace the method with: **return new string( input.Reverse() );**
4. Make the class **public**.
5. Compile the program and fix any errors that are produced.
6. Switch to the **Program.cs** file – add a new static method to the **Program** class. It should take no parameters and return **void**. Call it **Run**.
7. In this new method, we want to do the following:
   1. Create an **IStringProcessor** interface using our new **ReverseProcesor** type.
   2. Prompt the user to enter a string using the console.
   3. Read a string from the console.
   4. If the string is empty, exit the method. [**Hint**: the **String** class has a static **NullOrEmpty** method you can use to determine this].
   5. If the string is not empty, reverse it and print the result to the console and go back to step b.
   6. For fun, see if you can figure out how to change the foreground color of the text using the **Console** class so it is different during the prompt versus the results. Make sure you reset the color to its current value before exiting the method! [**Hint**: remember **try/finally**?]
8. Call the **Run** method at the end of the **Main** entry point (after you dump out the arguments), compile and run the program – debug it if you have issues.
   1. **Note**: In some cases, Visual Studio may give you an error about running a Class Library – this is because it has the wrong project selected for startup.

If you see an error message that looks like this:



Then you need to direct Visual Studio to start the ConsoleApp project instead – just right-click on the ConsoleApp project and select “**Set as Startup Project**”:



Another hint that this is the issue is that the currently selected “startup” project is in Bold – so if you see a Class Library in bold, then that is a hint you will need to select an actual application for debugging or running within Visual Studio.

## Lab Solution

Here is the lab solution – yours may look completely different and that is perfectly fine, but you can use this if you need a hint or two:

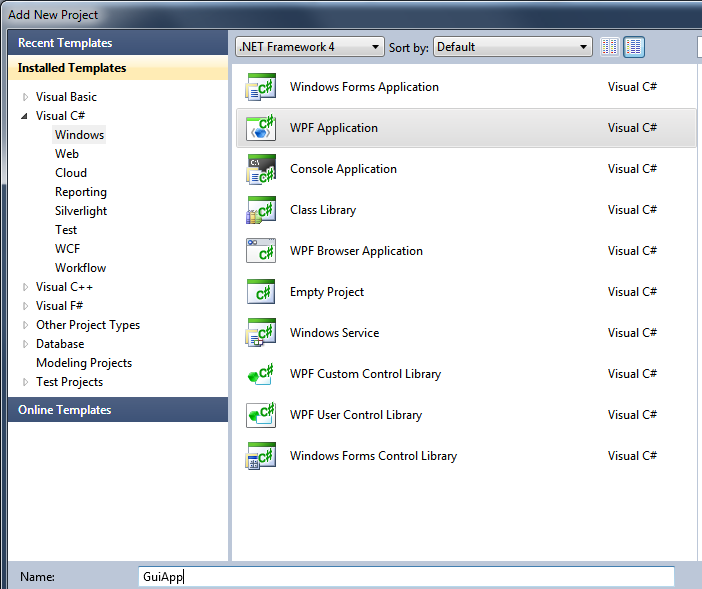
* 1. class Program
  2. {
  3. static void Main(string[] args)
  4. {
  5. foreach (string argument in args)
  6. {
  7. Console.WriteLine(argument);
  8. }
  9. Run();
  10. }
  11. private static void Run()
  12. {
  13. IStringProcessor processor = new ReverseProcessor();
  14. ConsoleColor originalColor = Console.ForegroundColor;
  15. try
  16. {
  17. while (true)
  18. {
  19. Console.ForegroundColor = ConsoleColor.Yellow;
  20. Console.Write("Enter Some Text: ");
  21. Console.ForegroundColor = originalColor;
  22. string input = Console.ReadLine();
  23. if (string.IsNullOrEmpty(input))
  24. return;
  25. string result = processor.Process(input);
  26. Console.ForegroundColor = ConsoleColor.Green;
  27. Console.WriteLine(result);
  28. }
  29. }
  30. finally
  31. {
  32. Console.ForegroundColor = originalColor;
  33. }
  34. }
  35. }

There is also a completed lab solution with all code located at [Task2.after\Lab1.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2001\Lab\Task2.after\Lab1.sln) if you would like to open a ready-to-run version.

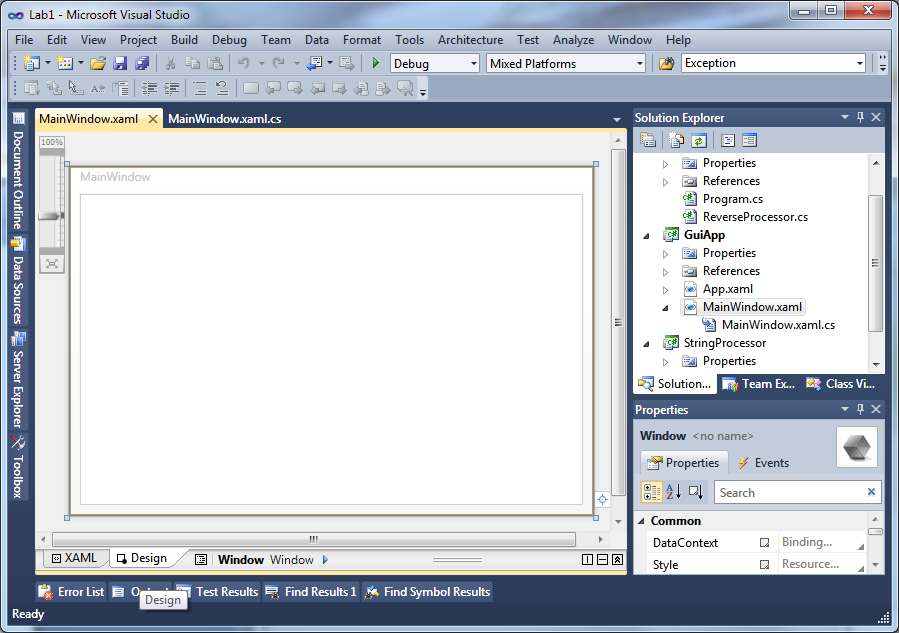
Task 3: Creating a GUI application

In this final task you will create a final application that will perform the same basic actions as your console version, but use the Windows Presentation Foundation (WPF) GUI features to create a true Windows application.

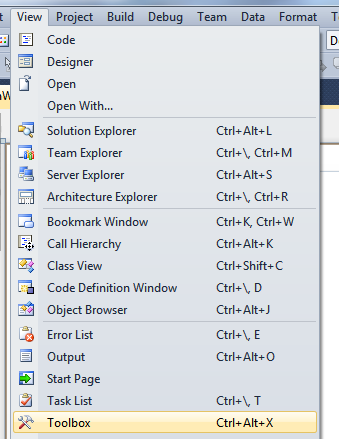
1. To start out, either open the final solution from Task 1 – located at [Task1.after\Lab1.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2001\Lab\Task1.after\Lab1.sln), or continue from task 1. You should have a single Console application project in your solution (named **ConsoleApp**).
2. Create a new project (**File 🡪 New 🡪 Project**, or right-click on the solution and select **Add 🡪 New Project…**)
3. Make sure you have C# highlighted, and select “WPF Application” as the project type from the list of templates. It is one of the first items – make sure you have the proper project as there are several “WPF related” templates here.
4. Name the new project “GuiApp” and place it alongside your current project.



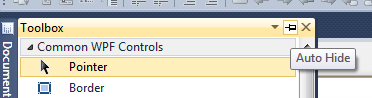
1. Make the GuiApp the active project by right-clicking on it and selecting “**Set as Startup Project**”. The project name will be bold when it is the startup project.
2. Visual Studio should have opened the **MainWindow.xaml** file – this is the markup file that represents the UI the user will interact with. We want to work with the designer version of this file, so if you see XML on the screen, then just click the “**Design**” button on the bottom of the Window to switch to the design view:



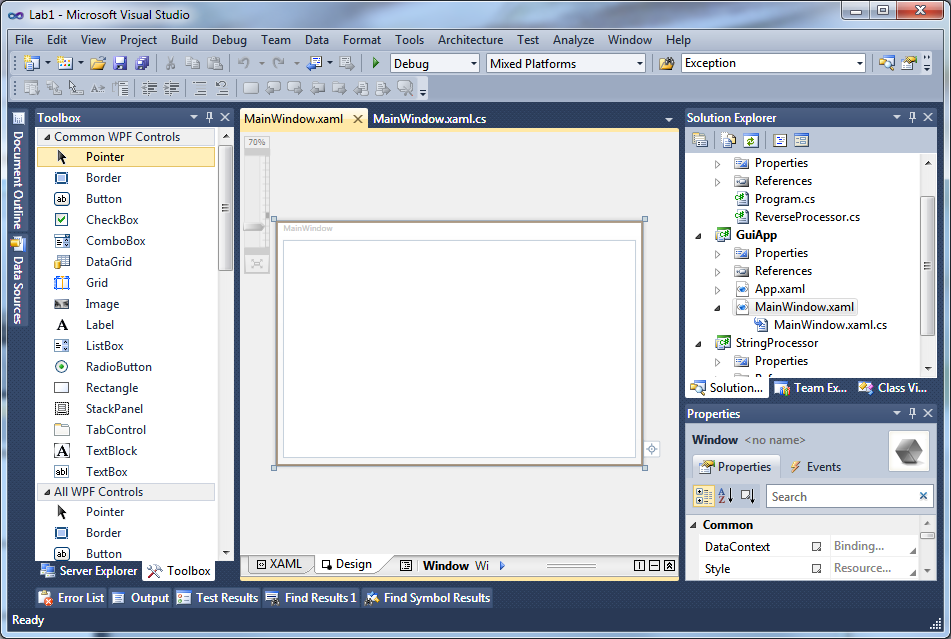
1. You should see a “**MainWindow**” white rectangle in the middle of the window – as shown above. Open the **Toolbox** – it should be on the left side of the screen as a tab. If you do not see it, just go to the **View** menu and choose **Toolbox** from there:



1. If you would like the toolbox to stay visible, you can click the pushpin in the corner of the window to dock it:

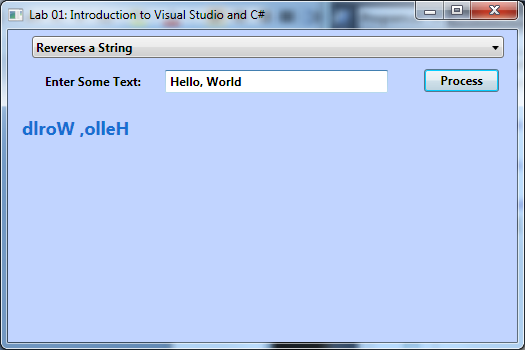


Your window should look something like this:

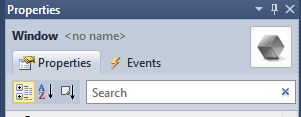


* 1. **Note**: You can control the scaling of the designer view by sliding the slider in the top left corner of the window as highlighted above. It allows you to zoom out to see the whole space if your monitor is not quite large enough to see it all, or zoom in to work on details of the view.

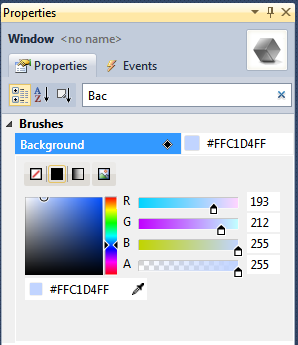
Our goal is to produce the following UI:



1. First, we will set the main window properties. Make sure the Window is selected in the designer. You should see the selected item in the properties window, or in the breadcrumb at the bottom of the design view, if it is not, try clicking on the design surface – specifically on the edge of the window so you select it. You can also use **CTRL+Click** to select between overlapping elements in the window.

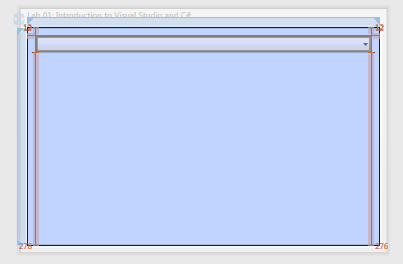
 

1. In the property explorer, find the **Title** property and set it to “Lab 01: Introduction to Visual Studio and C#”.
2. Find the Background property (try typing “Background” in the search field at the top of the property window to quickly locate it). Change the color to something you like.

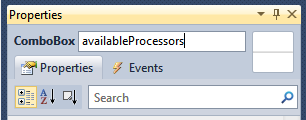


* 1. **Hint:** notice the “eyedropper” in the Brushes section above? If you click that, you can move the cursor around and the color box will change based on the current pixel under the mouse – this works even outside the Visual Studio application!

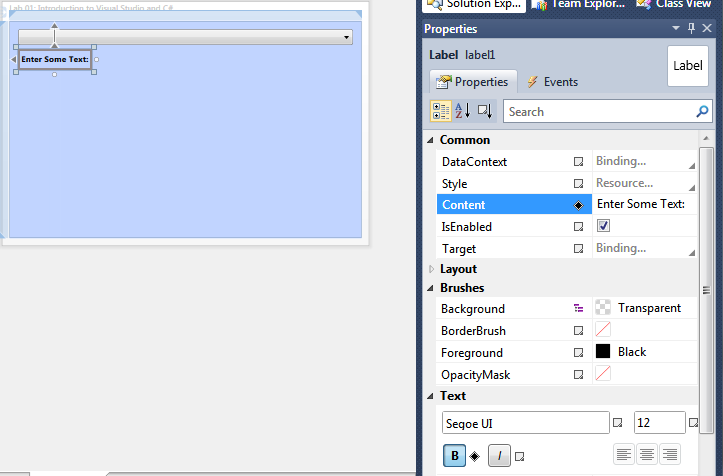
1. Now, we will add the controls. Use the above image to get a visual sense of what we are trying to create.
2. Drag a **ComboBox** element from the tool box onto the window – position it at the top of the window. You should see markers indicating margins and spacing so you can get it evenly sized. It will have drag corners to resize it.



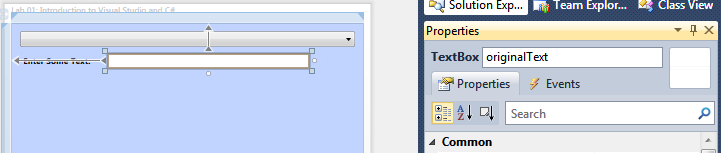
1. With the **ComboBox** still selected, switch to the **Property Explorer** and give it a name – the name is the text just to the right of the control type name in the property explorer. Name it “availableProcessors”. This name will create a **ComboBox** control object that we can then program against in our code behind when we start writing the code.



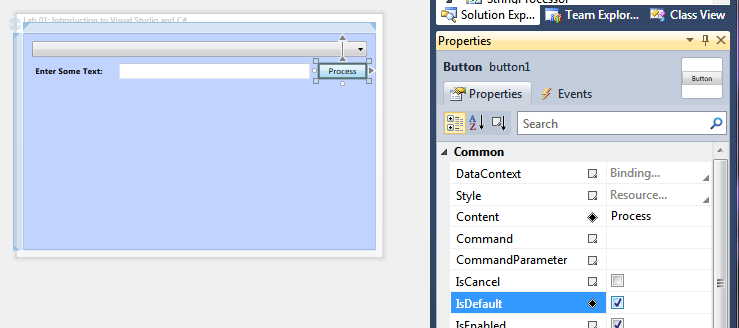
1. Next, drag a **Label** control over and position it below the **ComboBox** on the left side of the window. Switch to the **Property Explorer** and find the **Content** property – change it to “Enter Some Text:”. Also, expand the **Text** section of the **Property Explorer** and change the font to be Bold.



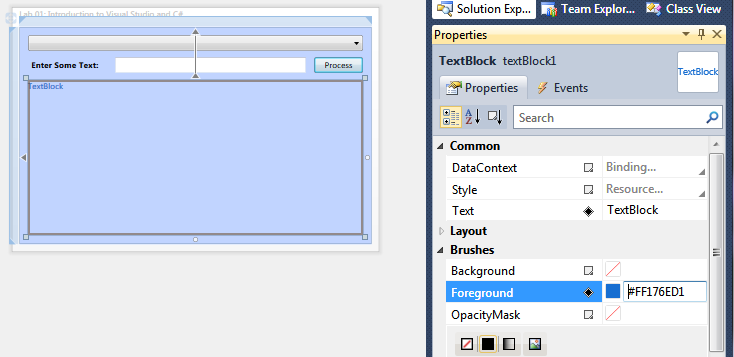
1. Next, drag over a **TextBox** control and position it to the right of the label. Resize it to take up ¾ of the remaining space. Switch to the **Property Explorer** and give it the name “originalText”.



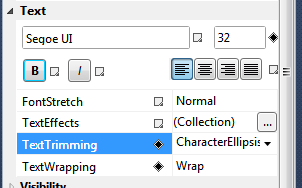
1. Next, drag a **Button** control and position it to the right of the **TextBox**. Change the **Content** property to “Process”. Also, check the **IsDefault** flag in the properties.



1. Finally, drag a **TextBlock** control and take up the remainder of the space. Change the **Foreground** property to a blue color:



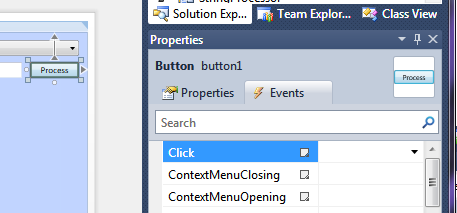
1. Give the TextBlock the name “resultText”. Expand the **Text** section of the properties for the **TextBlock**, set the font size to be something large – like 32, set the **TextTrimming** to be “CharacterEllipses” and the **TextWrapping** to “Wrap”. This will cause the text to wrap around the space, and then end with “…” if there isn’t enough room to display the entire text. This is a convenient feature of **TextBlock** and it is one of the main reasons to use it over a **Label**.



1. Now we are ready to begin programming the UI. Recall that we have named the UI elements with the following names:

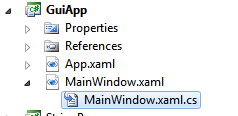
|  |  |  |
| --- | --- | --- |
| Name | Type | Purpose |
| availableProcessors | ComboBox | Will list the **IStringProcessor** types that are available. |
| originalText | TextBox | Input the text to transform from the user. |
| resultText | TextBlock | Resulting text from the processor output. |

We also have the button, we did not give it an explicit name (although it has one – button1), because we do not really need one. Instead, we simply want to know when the user **Clicks** the button. So, select the button object in the designer and then click the **Events** section of the **Property Explorer** – it is denoted by the lightning bolt icon:



1. Notice that “**Click**” is the default event – so it is highlighted by default. Type in a function name into the box – use **OnProcessText** and press **ENTER**. It will switch you to the code behind view – right in an event handler for this method. This is the method that will be invoked when you click the button and it’s handled automatically by the WPF framework.

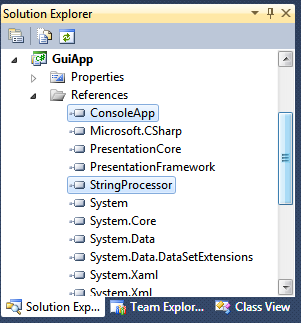
Code behind files are linked to the designer files – they share the same name, notice the designer file is named **MainView.xaml**, and the code behind file is **MainView.xaml.cs**. The solution explorer also denotes the relationship by having the code behind file as a child of the XAML file:



You can switch back and forth between the two files by either double-clicking on the file in the solution explorer, using the tabbed interface when both files are open, using the **View** menu, or use the F7 key to switch from designer to code behind view. The code behind file is where all your code will go that manages the UI, so it is here were we can populate the **ComboBox**, trigger some logic when the button is pressed, etc.

In this case, the event handler for the button press is where we want to get the processor, get the input text, run the processor on it and then set the results back to the **TextBlock** output. Before we can do that, we need to add a reference to our interface so we have access to the **IStringProcessor** interface.

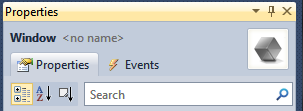
1. Right click on the references folder and select **Add Reference**.
2. Switch to the Projects tab – notice that the other two projects in this solution are listed, even though **ConsoleApp** is actually an executable. This is intentional and it points out a key difference between .NET and prior Windows executable formats -- .NET ships things as *assemblies* and it really doesn’t see a whole lot of difference between .EXE and .DLL formats – other than extension (there is a minor difference, but by and far they are the same). That means you can include classes contained in other .EXE assemblies as well as class libraries!
3. Add both references to your project by selecting both and clicking **OK**. You should see both listed in your references list for the **GuiApp** project.



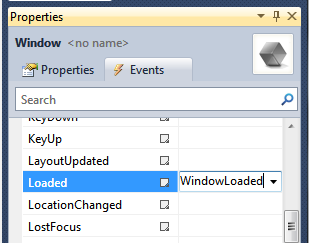
1. Switch back to the designer view for **MainWindow** (just double click on the XAML file in the solution explorer). Select the window either by clicking in the design surface, or you can use the breadcrumb along the bottom of the designer view – just click on the Window tag:



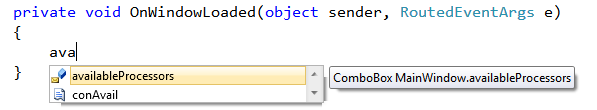
1. The selected object should now be Window in the Property Explorer:



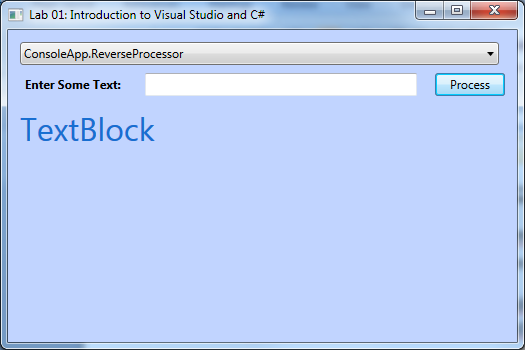
1. Switch to the **Events** tab (you may already be positioned there), the default event for the Window is the **Loaded** event. This indicates the window is initialized and ready to be displayed and it is often the best place to put your own initialization logic because all the controls have been created by this point.
2. Type “OnWindowLoaded” into the **Loaded** event box and press ENTER to generate the event handler.



1. In the code behind handler (**OnWindowLoaded**), we will populate the **ComboBox** with a processor. Remember we named the **ComboBox** “availableProcessors”, if you type that name you should see Intellisense begin completing the text:

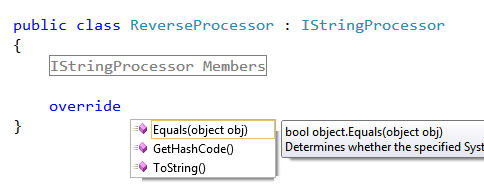


1. We want to populate the ComboBox.Items collection with processors, so use the following code to add the processor and then select it as the current choice:
   1. availableProcessors.Items.Add(new ReverseProcessor());
   2. availableProcessors.SelectedIndex = 0;
2. Make sure to add the appropriate **using** statement at the top [**hint**: remember how Visual Studio can help you out here].
   1. **Note:** If you can’t seem to see the class in Intellisense and you have the using statement there then it is very likely that the class is not public. The default visibility for a type is **internal**. One of the final steps in Lab 1 (step 27) had you change the visibility of the class to **public**. It will work without that but *only in that assembly*. So, if you missed that step, switch to the **ReverseProcessor** class and make it public so you can use the type here.
3. Run the program to see what you get:

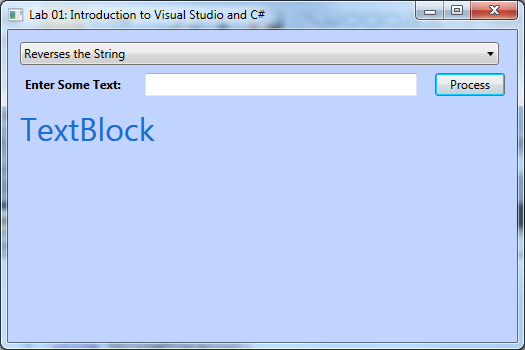


Hmm… well, it works – but the text is not quite right is it? The default behavior is to transform the type into a string to display in a control if it is not something naturally visual (such as a **TextBlock**). To do this, it is calling the **ToString** method on the type. By overriding that, we can provide a different text representation.

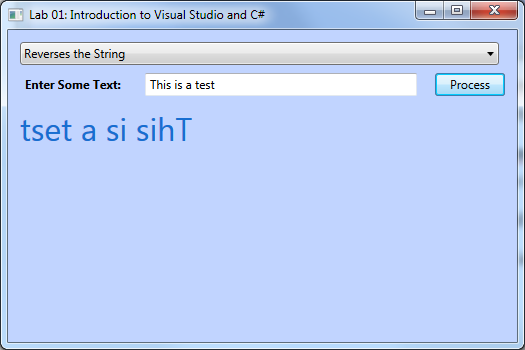
1. Switch to the **ReverseProcessor** class in ConsoleApp. Add an override method into the class for the **ToString** method and return a name for the processor. Try just typing the word **override** into the class, hit the space bar and see what Intellisense gives you. If you select one of the options it will generate the method for you!



1. In the **ToString** override, just return a sensible text string such as “Reverses the String”. Compile the solution and run the GuiApp again, you should see your output now:



1. Next, we will put our logic into place. Switch back to the **MainWindow.xaml.cs** code behind file and find the **OnProcessText** handler we added for the button click.
   1. Get the **SelectedItem** property from the **ComboBox**. This will be the selected processor – cast it to an **IStringProcessor** interface and store it into a file.
   2. Get the string from the input **TextBox** – it is stored in the **Text** property.
   3. Pass the string to the process and store the result into the **resultText.Text** property.
2. Here is an example implementation if you need a little help:
   1. private void OnProcessText(object sender, RoutedEventArgs e)
   2. {
   3. IStringProcessor processor = availableProcessors.SelectedItem
   4. as IStringProcessor;
   5. if (processor != null)
   6. {
   7. string input = originalText.Text;
   8. string result = processor.Process(input);
   9. resultText.Text = result;
   10. }
   11. }
3. Run the application – see how it works.



1. As a final exercise, try creating one or more additional **IStringProcessor** types and add them to the **ComboBox**. The examples shown in the final solution include the following, but you are encouraged to invent your own and explore the framework. Check out the final solution to see the implementations for:
   1. Change the string to Uppercase.
   2. Scramble the string (there is a **Random** class in .NET to generate random numbers from 0-n).
   3. Reverse only the letters in the words – not the string itself.
2. You can also set the initial value for the **resultText** to blank to remove the “TextBlock” text from the screen – you can do this by setting the **Text** property to blank in the designer.

## Lab Solution

The completed lab solution is available at [Task3.after\Lab1.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp3_MBF.V2.zip\MBF.V2\Module%2001\Lab\Task3.after\Lab1.sln). You are encouraged to look at the code for each of the sample processors to see more examples of built-in functions and features of .NET.