Welcome to *Get Going With F#*—A short book on learning how to use F# programming language for every day tasks. Each chapter in the book is a little mini-project where the reader is taught how to solve a particular type of problem using F#. The hope is that these projects contain enough information to get you started on your own projects.

This book assumes that you have programmed in at least one programming language for 2 to 3 years. Seasoned programmers with C#/C/C++ will have a special advantage. Unless otherwise stated, each chapter is stand-alone and can be read independently and out of order with the other ones, albeit, I do recommend that you read chapter 1 first even though you may already be familiar with F#.

- Chapter 1 is a very brief introduction to F# and functional programming. It ramps you up quickly so that you can start writing useful applications as soon as you have completed the chapter.
- Chapter 2 teaches you how to write a simple UI application for viewing fractals using WPF and FsXaml. At the end of the chapter you will have a fairly functional Julia and Mandelbrot set viewer.

F# is a functional-first programming language. That is, F# is a programming language which emphasizes the functional aspect of programming to a great extent. Much more so than other languages you may be familiar with such as C#, Visual Basic .NET, Python and JavaScript. You will already be quite familiar with functions: you met them in high school when you drew a plot of the function

$$y = x^2$$

using your trusty Sharp EL 9300 graphing calculator or your favorite plotting program on the WWW (e.g. Desmos at https://desmos.com/calculator). Functions as opposed to objects are the first-class citizens of functional programming languages like F#. They allow one to think about and solve problems using features prominent in functional programming such as immutability, pipelining, partial application, type-inference, and strong typing. (Note that these features are not unique to F#, they exist in other functional programming languages, and indeed in ordinary imperative programming languages such as C#, Visual Basic .NET, Python and JavaScript).

F# was created by Don Syme who is an Australian Computer Scientist and Principal Researcher at Microsoft Research, Cambridge, UK. He is still very active in the F# community and makes contributions to the F# language to this day (March 24th 2021). You could call him the Grandfather of F# or the Supreme

Leader of the F# community.

F# allows one to think about a particular programming problem or task in a clean and simple way. It has the clean simplicity of Python while retaining the performance and type safety found in languages such as C++ and C#. F# can be used to write small programs with specific goals set at the start to large systems with open-ended goals.

F# is fully open source and fully cross-platform. F# is integrated within Visual Studio and so can make full use of all the integrated development environment's (IDE's) features such as Intellisense, package management, documentation browser, etc. Alternatively, The Visual F# Tools can be downloaded from https://github.com/Microsoft/visualfsharp to develop F# programs without an IDE. On top of that, F# code can be run in Jupyter Notebooks hosted on Microsoft Azure Services.

I also use Visual Studio Code sometimes because it is cross-platform and has about a million different extensions from Markdown editing support to copy code/text with line numbers and everything else you can think of in-between.

As is traditional when learning most programming languages we will begin by writing a "Hello, World!" program. To do so first download and install the .NET SDK. You can search the Internet for instructions on how to download and install the .NET SDK. You are also free to choose whichever OS you prefer.

Open a command terminal and type in

```
dotnet new console -lang "F#" -o hello
```

`cd` into the directory hello which was created when you typed in the above command. Then, open the file `Program.fs` which was created when the directory was created. It should look something like:

```
   0:    // Learn more about F# at http://docs.microsoft.com/dotnet/
fsharp
   1:
   2:    open System
   3:
   4:    // Define a function to construct a message to print
   5:    let from whom =
   6:        sprintf "from %s" whom
   7:
```

```
 8:    [<EntryPoint>]
 9:    let main argv =
10:        let message = from "F#" // Call the function
11:        printfn "Hello world %s" message
12:        0 // return an integer exit code
```

Replace the contents of the file with just:

```
0:    printfn "Hello, World!"
```

Now go back to the terminal and type in:

```
dotnet run
```

Pretty cool! Right? One line of code is all that's needed to print out a message to the console and a one line command is all you need to build and run the executable that prints out the message. That's a lot better than C or C++ or C# and just as good as Python or any other scripting language.

I know what you're thinking!: There is something wrong here! Where is the function? There is a `printfn` function being called here which is taking a single string parameter as input and then outputting that string parameter to the console. But where is our function? Isn't F# supposed to be a functional-first programming language where nearly everything is a function? That is true. But in this case to make things easier to do, like simple scripting, the F# compiler is creating a "main" function for us which is then called by the .NET runtime. To make this more explicit we can wrap the "Hello, World" code inside of a function ourselves. So let's do that.

Change the code to the following:

```
0:    open System
1:
2:    [<EntryPoint>]
3:    let main(argv) =
4:      printfn "Hello, World!"
5:      0
```

Type in `dotnet run` to check that everything still works fine.

The code should be self-explanatory. Line 0 "open"s and makes the classes and objects in the .NET System library available for use by our little program. Line 2 sets the entry point for our program. Line 3 defines our function which takes in a list of strings as its one and only argument for the function. (Note that the type of the argument is not specified in the code. The F# compiler already knows what the type should be.) This parameter is not used in our program. Line 4 prints our "Hello, World!" message and finally line 5 returns the exit value. Notice that on line 5 we didn't need to write "return 0" as in some other languages that you may be familiar with. You just write down the actual return value and that's it!

Now that you've written and run "Hello, World!" in F# you should take a tour of F#. Start at the following URL http://docs.microsoft.com/en-us/dotnet/fsharp/tour. Once you feel ready to continue reading this book come back here and do the exercises in the following section.

1. Modify the "Hello, World!" program to ask the user for his/her name and print out "Hello <name of user>!" instead of "Hello, World!".
2. Write a function which will take two numbers and add them together and return answer.
3. Write a function which will compute the factorial of a number. You will need to research how to write recursive functions in F#.
4. Write a function which will compute the n^{th} term in the Fibonacci sequence.

You should also familiarize yourself with the `dotnet` command as you will be using it a lot. You can get help by typing in `dotnet -h` and also by searching and reading the Microsoft documentation online. Here are a few exercises that you can try after you've familiarized yourself with the `dotnet` command.

1. What is the default build configuration?
2. In which directory is the output stored?
3. What (SDK) command do you need to use to clean the output inside the build directory? Which files/directories still remain after you've cleaned the build directory?
4. What command do you need to type in to build a Release version of the "Hello, World!" application?
5. How do you run the Release version of the application.
6. How do you clean the Release build?

By now you should have a fairly good grasp of functional programming and F# in particular. You could stop reading this book now and continue your exploration of F# as needed for the particular set of projects that you are working on.

So what value does this book on F# have? Well, read on and you can decide for yourself whether the book fulfills any purpose and hence has any value. The remainder of this book contains mini-projects on various interesting topics from UIs to Neural Networks to 3D Computer Graphics. Each of the projects was chosen to illustrate how to solve a particular type of problem with F#. In the next chapter we will tackle the problem of writing UIs with F#. By the end of the chapter we will have a fairly functional fractal image generator for Julia and Mandlebrot sets. So without further delay let's get started!

In this chapter your wrote your first F# program!: A simple "Hello, World!" program. To do this you setup your system to write F# programs. You downloaded and installed the .NET SDK (if you didn't already have it installed) and then created a new console project. You then modified the code to print "Hello, World!" to the console.

You then took a tour of F# from around the web and returned to the book to complete some simple exercises. You also read some documentation on the `dotnet` command and played around with the command a little. You could say you are a beginner F# programmer, maybe even an intermediate F# programmer who is fairly competent at thinking about problems in a functional way. You are now ready to tackle what every programmer dreads and loves at the same time: UIs!

There are a number of paths available to programmers who decide to write UIs with F#:

1. Avalonia.FuncUI (Cross-Platform)
2. Elmish.WPF (Windows Only)
3. Use Windows Forms or WPF directly (Windows Only)

If you want to be cross-platform then option 1 is your best bet. Avalonia.FuncUI is fairly intuitive to use and will suit your purposes admirably, but at the time of writing (March 26th 2021) the documentation is so-so. So unless you like searching the web for answers and asking questions on Stack Exchange I don't recommend this route. Instead wait for 6 months and come back to Avalonia.FuncUI if you still need to write UIs in F#—the documentation will have improved by then.

Option 2, the Elmish.WPF route, is a Windows only route. It is a battle-tested UI framework with good documentation. It uses the model/view/update (MVU) pattern in its implementation. We will talk about the MVU pattern shortly.

Option 3 is also available for those who want to use their prior knowledge on Windows Forms or WPF directly in their apps on Windows. This is the route (with WPF) that we will take in this book as more readers will be aware of WPF than Avalonia.FuncUI or Elmish.WPF.

You can find out more information about the three options available at `http://fsharp.org/use/desktop-apps/`. Since we are going to be using WPF directly on Windows read this excellent article also to get started with WPF using F#: `https://www.c-sharpcorner.com/article/create-wpf-application-with-f-sharp-and-fsxaml/`.

Our fractal viewer application will use the Model/View/Update (MVU) architecture to implement its UI. Briefly, the model stores information about what is being viewed, the view displays the contents of the model, and the updater connects the model to the view and relays information from the model to the view when they need to communicate.

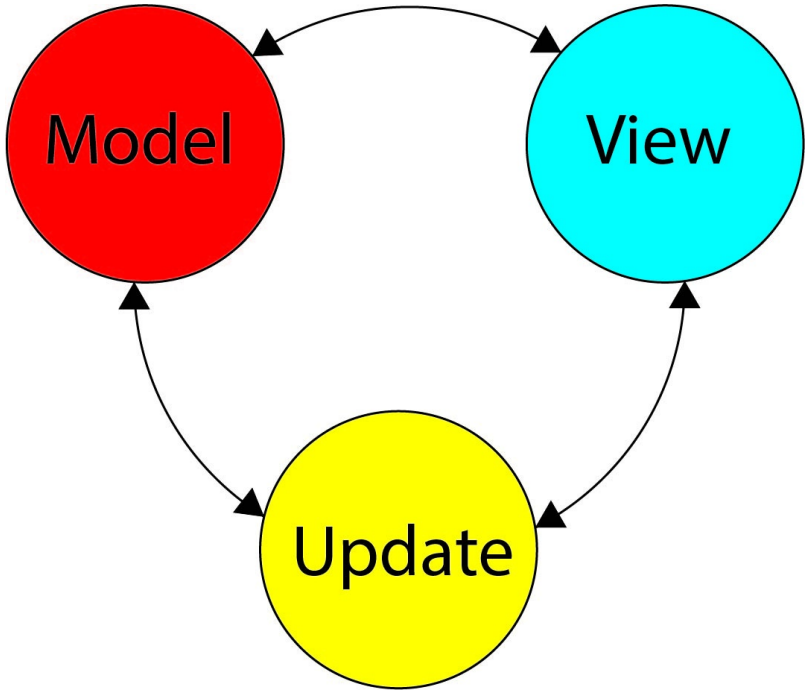
To make this more concrete, imagine you are writing a drawing application for colored pencils. The model would store information about the paper type—it's thickness, the color of the paper, the pattern of texture, the paper's reflectivity, and so on. It would also store information about the pencils available, their colors, whether they broke up easily when scraped across the paper, whether the pencil particles were soluble, and so on and on. The model would also store a representation of the paper and what had been drawn on the paper (perhaps as a multi-layer bitmap buffer).

The view is responsible for rendering the model. It would take whatever is stored in the model and render it using whatever rendering technology is available or desired for the view implementation. The view implementation could, for example, use Windows Presentation Foundation (WPF) to render the contents of the pencil drawing model. Or the view could use one of those fancy new rendering APIs like DirectX 12 or Vulkan to render the pencil drawing. Or it might just use plain and simple good old OpenGL. Or the view implementation could be something like a colored PDF file. Or it could even be something more exotic like a computer controlled robotic hand which creates the drawing with real colored pencils. You get the idea!

Finally, the update part of the MVU architecture connects the view to the model by passing messages to the model. Whenever the model needs to be changed, for example when a user moves the mouse across the table with the left mouse button

held down indicating that a new stroke needs to be drawn, the updater intercepts these mouse interactions and modifies the model accordingly. The view is then notified about the new model state causing it to re-render the contents of the model.

So you can think of the model, view, and update implementations as forming a loop or circle. The model affects the view which affects the update which affects the model and so on.



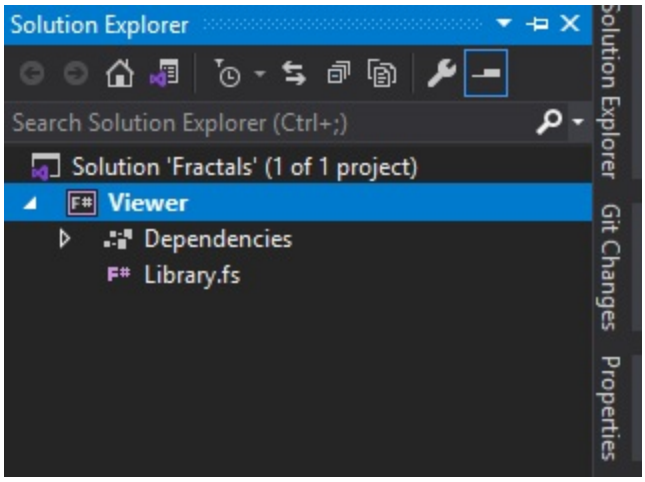
Let's take what we've learned so far to create our Fractals viewer using WPF and FsXaml for our UI. Since we are going to be working in Windows in this chapter we will use Visual Studio Community Edition for our development. Visual Studio Community Edition is free and you can download and install it from `https://visualstudio.microsoft.com/vs/community/`. Install the F# tools when you install Visual Studio Community Edition. Take some time to learn how to use Visual Studio, especially features such as debugging and performance profiling you haven't used them before.

When you are are ready, type in the following commands into a Powershell command prompt to create the skeleton of the Fractals viewer project:

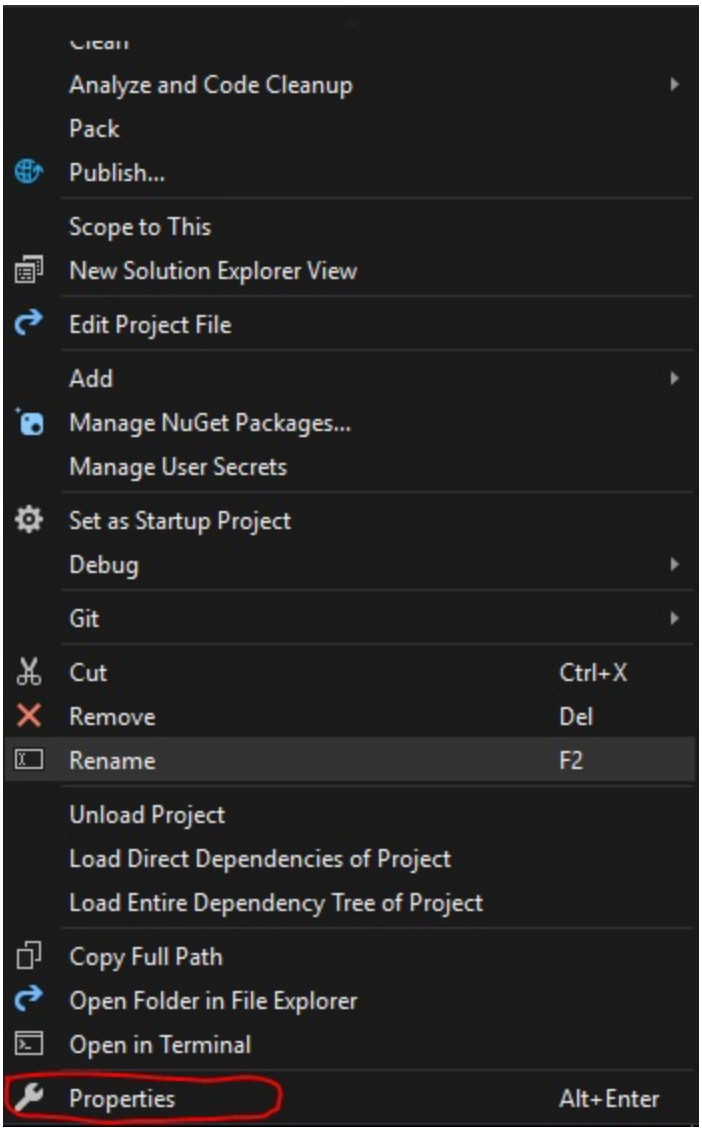
```
1: dotnet new sln -n Fractals
2: dotnet new classlib -lang "F#" -n Viewer
3: dotnet sln add Viewer/Viewer.fsproj
```

A quick explanation of what each line means: In line 1 we create a new Visual Studio solution file called Fractals. In line 2 we create a new F# based class library called Viewer. The files associated with this Viewer are stored in the sub-directory Viewer. Finally, on line 3 the Viewer class library is added to the solution.

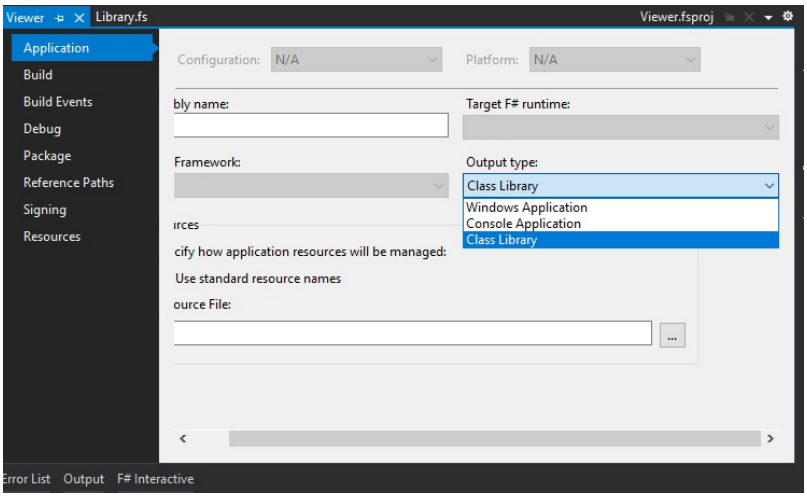
Now open up Visual Studio and open the solution file you just created. Right-click on the Viewer project in the Solution Explorer:



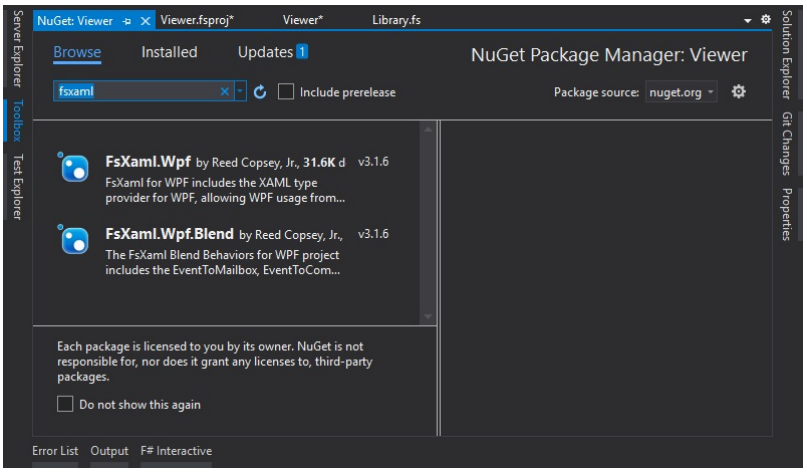
and select *Properties* in the context menu that appears:



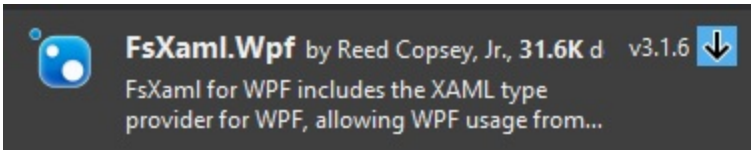
In the Viewer properties page, in the *Application* tab select *Windows Application* instead of *Class Library* from the *Output type:* menu.



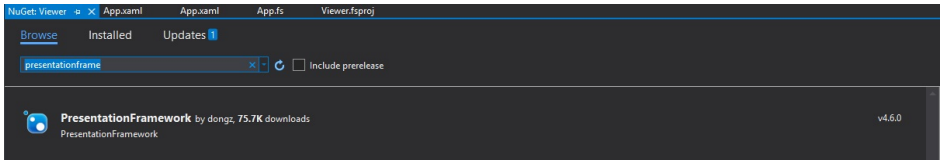
Now right-click on the Viewer project again and select Manage NuGet Packages … from the context menu that appears. In the NuGet package manager, type in FsXaml in the *Browse* box:



Select and install FsXaml.Wpf by pressing the downwards pointing arrow which appears when you hover your mouse over the FsXaml.Wpf button:

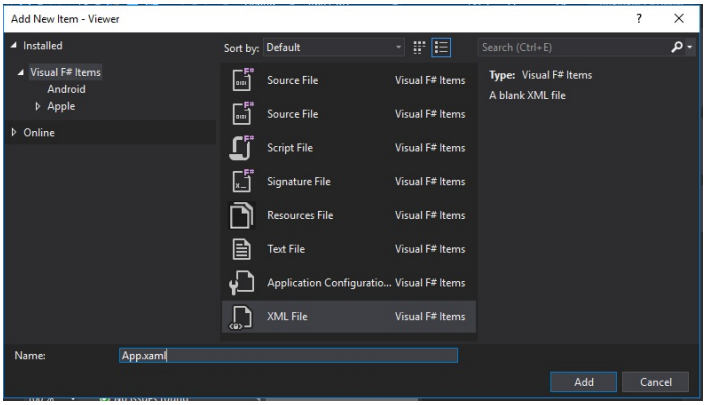


Similarly add a reference to PresentationFramework

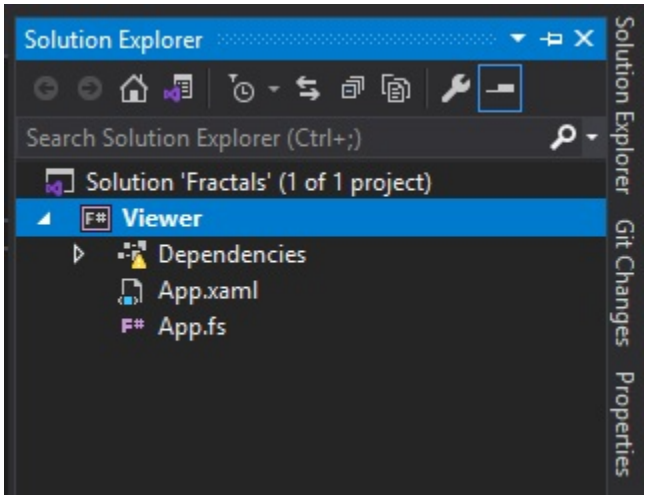


We are going to follow the instructions at `https://www.c-sharpcorner.com/article/create-wpf-application-with-f-sharp-and-fsxaml/`. First Rename Library.fs to App.fs.

Then add a new XML file to the project and rename it App.xaml: Right click on the *Viewer* project in the Solution Explorer and select *Add* > *New Item*... Call the new file App.xaml



Your solution explorer should now look like:



Replace the code in App.xaml with:

```
   1:    <Application
   2:        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
   3:        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:        StartupUri="MainWindow.xaml">
   5:
   6:      <Application.Resources>
   7:      </Application.Resources>
   8:
   9:    </Application>
```

Then replace the contents of App.fs with the following:

```
   1:    module App
   2:
   3:    open System
   4:    open FsXaml
   5:
   6:    type App = XAML<"App.xaml">
   7:
   8:    [<EntryPoint;STAThread>]
   9:    let main argv =
  10:      App().Run()
```

As mentioned in the blog, line 6 generates a new App type.

Similar to how you created App.xaml, create a new file called MainWindow.xaml and add the following contents to the file:

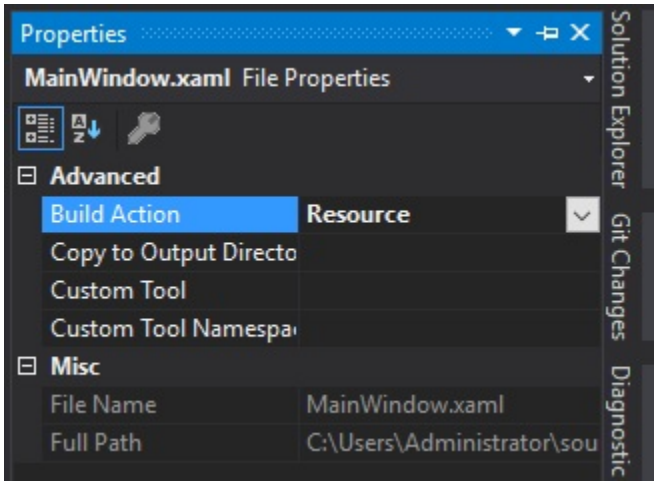
```
   1:    <Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
   2:        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   3:        xmlns:local="clr-namespace:Views;assembly=FsXamlApp"
   4:        xmlns:fsxaml="http://github.com/fsprojects/FsXaml"
   5:        Title="F# Windows App" Height="300" Width="400">
   6:
   7:      <Grid>
   8:        <TextBlock Text="Hello F# from WPF!"
   9:                   HorizontalAlignment="Center"
  10:                   VerticalAlignment="Center" />
```

```
11:       </Grid>
12:     </Window>
13:
```

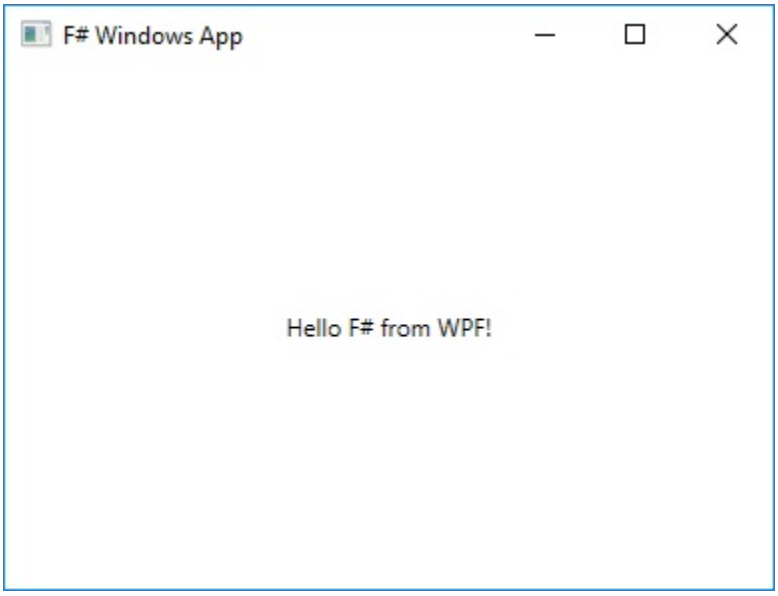
Next add the codebehind for the file in a new file called MainWindow.xaml.fs:

```
1:     namespace FsXamlApp
2:     open FsXaml
3:
4:     type MainWindow = XAML<"MainWindow.xaml">
```

Now right-click on the MainWindow.xaml file in Solution Explorer and change the BuildAction to resource as shown below. Do the same for App.xaml.



You should now be able to run the skeleton application and the following window should be shown when you do so.



Before we continue any further let's make a small detour and ask ourselves "What is a fractal?" and specifically "What is a Julia Set" and "What is the Mandlebrot Set?" You may skip this section and the subsections contained therein if you are a Mathematician or your are already familiar with fractals.

This section collects useful tricks, cheats, and patterns for using and programming with F#. If you found any particularly useful patterns that you like, please send them to me at `raghavendra.chandrashekara@outlook.com`.

Thanks!