Walkthrough: Seeding the REPL from a WPF Project to Explore Snippets

With the new **C# Interactive window**, you get immediate feedback on what an expression will return or what an API call does. The Interactive window is much like the Immediate window, but the Interactive window has many improvements such as intellisense features and the ability to redefine functions and classes. The interactive window builds execution context by evaluating input code snippets, not by attaching to a debugee process.

After entering a code snippet at the REPL prompt, the code simply executes right away. You can enter statements and expressions, as well as class and function definitions. You do not need to create project, define a class, define a Main, write your expression in a call to Console.WriteLine(), and use a bogus Console.ReadLine to keep the cmd.exe window alive after you hit F5. When you type each line of code that is a complete expression or construct and then press Enter, the code executes. If the entered code is incomplete when you press Enter, it does not execute, and you can continue entering code.

The Interactive window is a read-eval-print loop (or REPL), <http://en.wikipedia.org/wiki/REPL> . As explained, REPLs are productivity enhancers that have long been the purview of dynamic and functional languages. Some explicitly typed static languages have had REPL offerings, such as SuperCede 2.0 for Java or Mono's C# REPL. The Roslyn REPL for C# brings all the goodness you associate with VS such as completion, code coloring, Quick Fixes, etc., to the REPL experience.

To frame this walkthrough, imagine you have already roughed out a UI (imagine various buttons, editing area, etc., sitting around the tan square that is in the UI now). You need to write the code to draw a grid on the tan square that has the number of lines as input in some part of the UI (left out for simplicity of demo). You have already figured out how to define a WPF Grid and draw individual lines from code, but you have to divide up the area evenly and leave a border too.

To use this walkthrough, you must first install Visual Studio 2012, and then install the Roslyn Community Technology Preview (CTP). The CTP is at <http://go.microsoft.com/fwlink/?LinkId=228400> .

This walkthrough shows seeding the REPL from a project context and then developing some snippets to programmatically add objects to a WPF UI. It shows the following:

* Launching the C# Interactive Window (or REPL) and seeding the execution context from a project
* Calling into an application to poke around at live running code
* Keeping REPL interaction live while running WPF app (dispatching input to UI thread of app)
* #load of helpers
* Using directives
* History commands
* Enter to execute if input is complete
* Colorization, completion, signature help (param tips)
* Expression evaluation
* Multi-line input
* Multi-line history with editing
* Load file of top-level helper functions

**Walkthrough Steps …**

**1.** Open Visual Studio with the Roslyn CTP installed.

**2.** Open the wpfseedrepl.sln samples project, which installed to your documents folder under "Microsoft Roslyn CTP - September\csharp\wpfseedrepl\wpfseedrepl.csproj".

**3.** Right click on the WpfSeedRepl project node and choose Reset C# Interactive from Project. If you are presented with a dialog to save the solution, go ahead and save. This command displays the REPL tool window.

**4.** If you did not do the introductory walkthrough for the REPL, it is best to drag the REPL window to the document bay and dock the interactive window at the bottom of the document bay. It is common to work in an editor buffer and the REPL, switching back and forth for entering and saving code snippets, particularly if you are writing a script.

**5.** You may notice the tool window's toolbar has command icons for clearing the Interactive window's contents and resetting the execution context; there is no button to interrupt the currently executing code submission at this time. If you enter an infinite loop while playing around, you can use the button to reset the REPL. You will reset the execution context completely and will need to restart the walkthrough.

You will see the initial execution context in the REPL includes the references from the project and a 'using' for the project's default namespace

Resetting execution engine

> #r "System"

#r "System.Data"

#r "System.Xml"

#r "Microsoft.CSharp"

#r "System.Core"

#r "System.Xml.Linq"

#r "System.Data.DataSetExtensions"

#r "System.Xaml"

#r "WindowsBase"

#r "PresentationCore"

#r "PresentationFramework"

#r "wpfseed.exe"

using WpfSeedRepl;

**6.** To work within the project context, there needs to be more 'using' context. Enter the following (copied from the top of MainWindow.xaml.cs). If you enter these manually, completion will help you, and you can use Shift-Enter between the 'using' lines and one final Enter at the end to execute. Pressing Enter each time is fine too.

> using System;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

**7.** Now you are ready to launch the app by instantiating the MainWindow. You do NOT do this by invoking App.Main(). The REPL process is not your WPF application's process. Notice above in the #r's, the WPF application's exe was loaded as an assembly. Therefore, calling App.Main() will fail to find your xaml resources. To launch the application, enter the following lines, pressing Enter to execute the input.

> var win = new MainWindow();

win.Show();

You can skip this and alt-tab to the window to see progress, but it is also helpful to keep the window on top:

> win.Topmost = true;

**8.** Due to how the REPL process works today, the interactive code you type in the REPL cannot access non-public definitions in your project. To work around that for now, there are some helper access definitions at the end of the MainWindow class, all named with the "repl" prefix. Invoke the ReplSetup function to initialize the helpers:

> win.ReplSetup();

**9.** You need to figure out how to draw the right number of lines evenly spaced. You have already figured out how to define a grid and draw lines from code, but you have to divide up the area and have border space around the grid. You know you'll need helper functions for defining rows and columns. The helpers take a unit of proportion for a given row or column, and they define the column width or row height to be auto-sizing with the relative proportion passed into the function. Let's define those.

> Func<int, ColumnDefinition> defineColumn = (proportion) => {

var col\_def = new ColumnDefinition();

col\_def.Width = new GridLength(proportion, GridUnitType.Star);

return col\_def;

};

> Func<int, RowDefinition> defineRow = (proportion) => {

var row\_def = new RowDefinition();

row\_def.Height = new GridLength(proportion, GridUnitType.Star);

return row\_def;

};

**10.** Define a few helper "constants" for our learning exercise. These represent the size of the grid in number of lines needed in each dimension, how many units of space to allot for border space, line width in pixels, etc.

> int size = 4;

int border = 1;

int size\_w\_borders = size + (border \* 2);

const int LINE\_WIDTH = 2;

Just to save some typing, define "g" to be "win.ReplGrid":

> var g = win.ReplGrid;

**11.** Create the grid to hold the lines you want to draw. The grid includes two border rows and two border columns (hence, size\_w\_borders):

> for (int i = 0; i < size\_w\_borders; i++) {

g.ColumnDefinitions.Add(defineColumn(1));

g.RowDefinitions.Add(defineRow(1));

}

It is helpful to turn on visible grid lines:

> g.ShowGridLines = true;

**12.** Now you need two more helper functions to draw lines. Again, you've already played around and figured out how to draw a line, but you still need to figure out how to place the lines. These functions draw a horizontal or vertical line that is size in length, starting one border unit offset in the containing grid.

Enter these definitions in the REPL:

> Rectangle MakeHorizontalLine (int loc, int size,

VerticalAlignment alignment) {

var hrect = new Rectangle();

Grid.SetRow(hrect, loc);

Grid.SetColumn(hrect, border); // Skip border

Grid.SetColumnSpan(hrect, size);

hrect.Height = LINE\_WIDTH;

hrect.Fill = new SolidColorBrush(Colors.Black);

hrect.VerticalAlignment = alignment;

return hrect;

}

Rectangle MakeVerticalLine (int loc, int size,

HorizontalAlignment alignment) {

var vrect = new Rectangle();

Grid.SetRow(vrect, border); // Skip border

Grid.SetColumn(vrect, loc);

Grid.SetRowSpan(vrect, size);

vrect.Width = LINE\_WIDTH;

vrect.Fill = new SolidColorBrush(Colors.Black);

vrect.HorizontalAlignment = alignment;

return vrect;

}

**13.** Now you can draw the lines to see if our grid looks ok:

> for (int i = 1; i < size\_w\_borders - 1; i++)

{

g.Children.Add(MakeHorizontalLine(

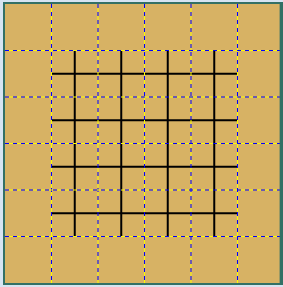
i, size, VerticalAlignment.Center));

g.Children.Add(MakeVerticalLine(

i, size, HorizontalAlignment.Center));

}

However, you can see that the lines appear to stick out incorrectly:



**14.** Imagine that you've sat back, thought about this some, and you realized you can't just draw the lines starting in the center of a grid cell, and you can't draw them all flush left or flush right and get a nicely centered board. You realize the way to use WPF is to divide in half the rows and columns containing the outer-most lines. Then you can draw the outer-most lines flush left, top, right, and bottom in the appropriate grid cells, and place all the other lines centered in the regularly sized grid cells.

You need to remove lines, rows, and column definitions to start over:

> g.Children.Clear();

g.ColumnDefinitions.Clear();

g.RowDefinitions.Clear()

**15.** Enter the following code to set up the rows and columns where they are all uniform width or height, but two rows and two columns are split in half to get the lines to draw correctly. Obviously, you can omit comments, and if you like, you can select the commented out code in setup.csx and use ctrl-e ctrl-enter [BUG in CTP: use c-enter instead] to Execute in Interactive.

> g.ColumnDefinitions.Add(defineColumn(2)); // border is uniform width

// split first column so line ends in middle of cell

g.ColumnDefinitions.Add(defineColumn(1));

g.ColumnDefinitions.Add(defineColumn(1));

// most columns are uniform width

for (int i = 0; i < size - 2; i++) {

g.ColumnDefinitions.Add(defineColumn(2));

}

// split last column so line ends in middle of cell

g.ColumnDefinitions.Add(defineColumn(1));

g.ColumnDefinitions.Add(defineColumn(1));

g.ColumnDefinitions.Add(defineColumn(2)); // border space is uniform width

g.RowDefinitions.Add(defineRow(2)); // border space is uniform height

// split first row so line ends in middle of cell

g.RowDefinitions.Add(defineRow(1));

g.RowDefinitions.Add(defineRow(1));

// most rows are uniform width

for (int i = 0; i < size - 2; i++) {

g.RowDefinitions.Add(defineRow(2));

}

// split last row so line ends in middle of cell

g.RowDefinitions.Add(defineRow(1));

g.RowDefinitions.Add(defineRow(1));

// border space is uniform height

g.RowDefinitions.Add(defineRow(2));

**16.** Imagine while sitting back and thinking about the problem, you also figured out that to use our half rows and columns correctly, you need to place the outside lines differently than the middle lines. Enter the following code to draw new lines (or select it in the setup.csx file and use ctrl-e ctrl-enter to execute the selection):

> g.Children.Add(MakeHorizontalLine(border + 1, size,

VerticalAlignment.Top));

g.Children.Add(MakeVerticalLine(border + 1, size,

HorizontalAlignment.Left));

for (int i = border + 2; i < size + 1; i++) {

g.Children.Add(MakeHorizontalLine(i, size,

VerticalAlignment.Center));

g.Children.Add(MakeVerticalLine(i, size,

HorizontalAlignment.Center));

}

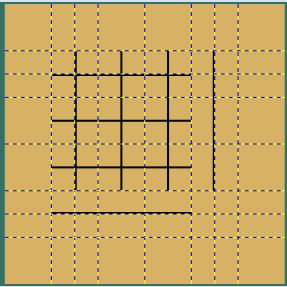
g.Children.Add(MakeHorizontalLine(size + 1, size,

VerticalAlignment.Bottom));

g.Children.Add(MakeVerticalLine(size + 1, size,

HorizontalAlignment.Right));

You can see it still not quite right:



**17.** You can see that the lines are all starting right up against the border row or column. The code defined the half rows and columns to move the line ends to what would look like the middle of the larger or uniform row and column size. Looking at the functions that draw lines, you can see that the code starts to draw lines offset by only one row or column (using the variable 'border'). You need to redefine these functions to start lines "border + 1" cells in from the edge of the board so that you’re using the half row and half column defined to get the line ends to meet on the edge lines.

Scroll back in the REPL buffer until you see the two functions you entered that define MakeHorizontalLine and MakeVerticalLine. With the caret in this old input block, use ctrl-enter to copy this previous input to the current pending input. Use the arrow keys or the mouse to change the two lines that reference 'border' to add one to 'border', and then press ctrl-enter to execute:

> private Rectangle MakeHorizontalLine(int loc, int size,

VerticalAlignment alignment) {

var hrect = new Rectangle();

Grid.SetRow(hrect, loc);

Grid.SetColumn(hrect, border + 1); // Skip border and half column

Grid.SetColumnSpan(hrect, size);

hrect.Height = LINE\_WIDTH;

hrect.Fill = new SolidColorBrush(Colors.Black);

hrect.VerticalAlignment = alignment;

return hrect;

}

private Rectangle MakeVerticalLine(int loc, int size,

HorizontalAlignment alignment) {

var vrect = new Rectangle();

Grid.SetRow(vrect, border + 1); // Skip border and half row

Grid.SetColumn(vrect, loc);

Grid.SetRowSpan(vrect, size);

vrect.Width = LINE\_WIDTH;

vrect.Fill = new SolidColorBrush(Colors.Black);

vrect.HorizontalAlignment = alignment;

return vrect;

}

**18.** Let's clear the lines again with the following code. You can get back this line of code with alt-uparrow. You could also scroll in the REPL buffer until you see this as earlier input and then use ctrl-enter with the caret inside the previous input. If you use one of these history tricks, and you entered all the lines of step 14 in one submissions, you will need to delete two of the lines so that your input looks like the following:

> g.Children.Clear();

**19.** Now use the history command (alt-uparrow) to get back to the following input and hit enter to evaluate it:

> g.Children.Add(MakeHorizontalLine(border + 1, size,

VerticalAlignment.Top));

g.Children.Add(MakeVerticalLine(border + 1, size,

HorizontalAlignment.Left));

for (int i = border + 2; i < size + 1; i++) {

g.Children.Add(MakeHorizontalLine(i, size,

VerticalAlignment.Center));

g.Children.Add(MakeVerticalLine(i, size,

HorizontalAlignment.Center));

}

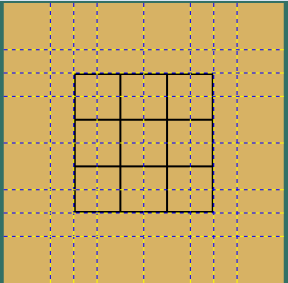
g.Children.Add(MakeHorizontalLine(size + 1, size,

VerticalAlignment.Bottom));

g.Children.Add(MakeVerticalLine(size + 1, size,

HorizontalAlignment.Right));

Now we can see that we have a centered set of lines that all end crisply on another line:



**20.** Now that we've figured out how to draw the lines the way we want, we can take the working snippets of code and assemble some functions in our program to draw our UI based on an input size. Let's clear the lines and grid again with the following code, which you can get back with alt-uparrow or scrolling in the REPL buffer until you see it and then using ctrl-enter with the caret inside the previous input:

> g.Children.Clear();

> g.ColumnDefinitions.Clear();

g.RowDefinitions.Clear()

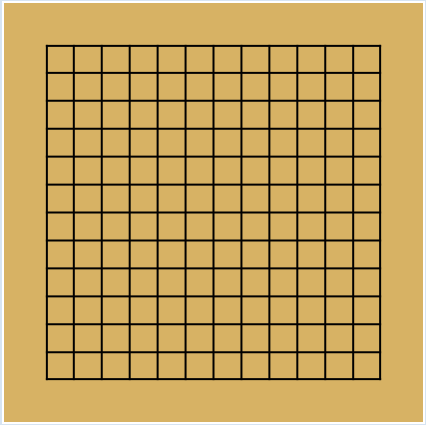
**21.** Let's also hide those grid lines:

> g.ShowGridLines = false;

Now let's execute the function already written in the project to draw a full a grid 13x13.

> win.ReplFillLinesGrid()

Then we see :



You're done. Enjoy using the REPL and please provide feedback!