Exercise: Creating an Object Oriented System for Drawing Shapes using C#

version 20.06.17

Contents

Introduction

This exercise is meant to practise fundamental object oriented concepts. A set of related classes is created regarding several simple mathematical 2D shapes. In a working program objects of these classes will be created and used to make concepts more tangible and give insight into the inner workings of these matters.

Part 1: Preparing the Solution and Projects

Create a new Console App Project inside an automatically created Solution

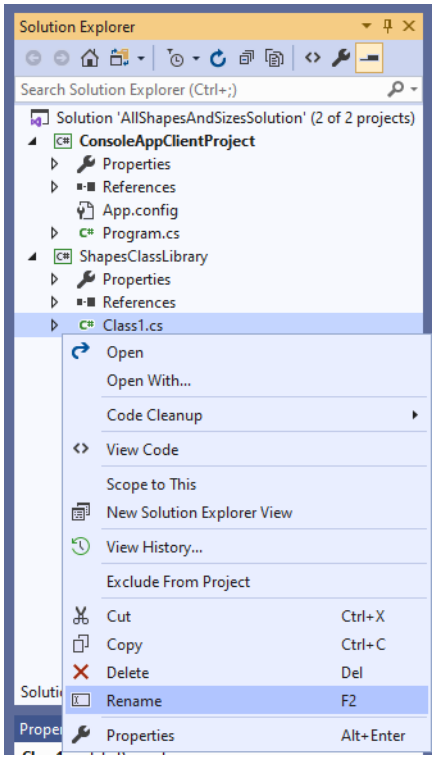
1. Open Visual Studio and select [Create a new project]
2. Make sure to change the drop down filters to the following settings:
    - [All languages] → C#
    - [All platforms] → Windows
    - [All project types] → Console
3. Now select "Console App (.NET Framework) and click [Next]
4. Enter for **Project name**: *ConsoleAppClientProject*
5. Choose a **Location** that suits you…
6. Enter for **Solution name**: *AllShapesAndSizesSolution*
7. Make sure checkbox ☐ **Place solution and project in the same directory** is not checked
8. For **Framework** select the highest available number in the drowdown list and click [Create]

Add a Class Library Project to the current Solution

1. In the Visual Studio top menu select File > Add > New Project…
2. In the now appearing dialog entitled "Add a new project" change the most right drop down filter from [Console] to [Desktop]
3. Scroll down in the results to find and select "Class Library (.NET Framework). You could also use the search text box and start typing the word *class* to limit the options in the listed results.
4. Enter for **Project name**: *ShapesClassLibrary* and click [Create]

Part 2: Implementing the Classes

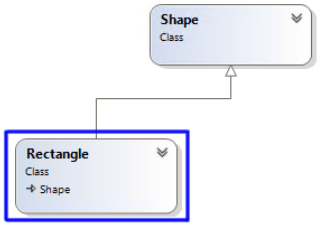
Create the Base Class



1. To give the base class and its file a proper name in one go, in the Solution Explorer panel *right-click* the file *Class1.cs* and select *Rename* as shown in the picture.
2. Change the name to **Shape.cs** and if asked to rename all references click [Yes].
3. Make sure your **Shape** class contains the following:
    a. two *private variables* of type double, one named **length** and one named **width**
    b. one *public constructor* taking length and width as parameters and when called, storing their values in the belonging variables created in step a.
    c. two public *getter only* properties returning the stored length and width, respectively.
    d. *override* the **ToString** method and make sure it returns a string telling something like 'I am a shape with measures *so-and-so*…'
    e. two public *virtual* methods called **CalculatePerimeter** and **CalculateArea** both returning a *double* with a hardcoded value of -1.0 (*minus* 1.0 that is)

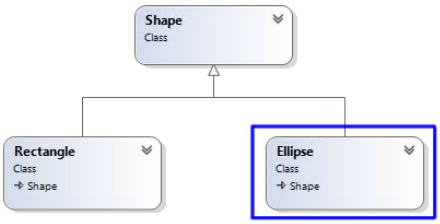
Do NOT forget to **save** your work once in a while…

Add the first Subclass (Rectangle)



1. Make sure in the Solution Explorer one of the files belonging to *ShapesClassLibrary* is selected.
2. In the Visual Studio top menu select Project > Add Class…
3. **Name** the class Rectangle and click [Add]
4. IMPORTANT: in the newly created class file make sure the class is *publicly accessible*, so it should read:
   `public class Rectangle`
5. Make sure your **Rectangle** class contains the following characteristics and code:
    a. destined to be a sub/child/derived class of Shape, it should inherit from class Shape
    b. one *public constructor* taking length and width as parameters, passing these on to its parent class (without any implementation of its own) by using this syntax at the end:
       `: base(length, width) { }`
    c. *override* the **ToString** method and make sure it returns a string telling something like 'I am a rectangle with measures *so-and-so*…'
    d. two public *override* methods called **CalculatePerimeter** and **CalculateArea** both returning a *double* but this time returning actual calculated rectangle perimeter and area values respectively.

Add the second Subclass (Ellipse)

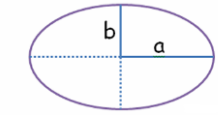


1. Add another class to the ShapesClassLibrary by following the first 4 steps of the previous section, this time naming the new class **Ellipse**
2. Follow the 5th step of the previous section as well, changing the details regarding *rectangle* to *ellipse* accordingly. Formulas are shown next:

The formula to *approximate* the circumference of an ellipse: $2\pi\sqrt{\frac{a^2+b^2}{2}}$

Calculating the area of an ellipse is fortunatelly a lot simpler: πab

where a and b are the lengths of the semi-major and semi-minor axes, respectively as shown below:



Explanations and calculators to check the programmed results can be found here:

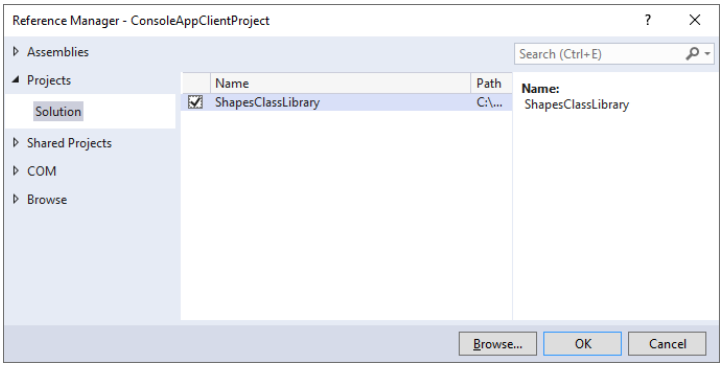
b. https://miniwebtool.com/ellipse-circumference-calculator/
c. https://miniwebtool.com/area-of-an-ellipse-calculator/

Part 3: Implementing the Console App for the First Test Run

Enable the Class Library to be used by the Console App

1. In the Solution Explorer select ConsoleAppClientProject.
2. In the Visual Studio top menu select Project > Add Reference…
3. Open Projects > Solution and check ☑ ShapesClassLibrary as shown in the picture. Click [OK].
4. From the Solution Explorer open file *Program.cs*
5. To be able to use the classes in the Class Library without always typing their namespace(s) in front of them, on a new line after the last using statement enter:

```
using ShapesClassLibrary;
```



Instantiate Objects from the First Classes

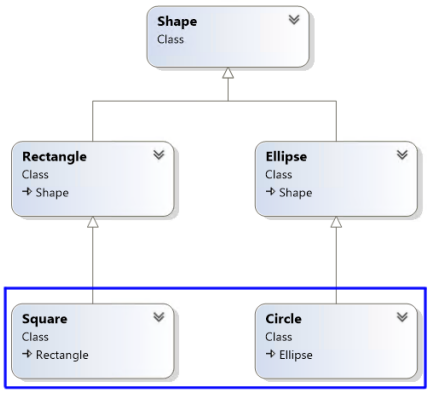
1. In file *Program.cs* write code in the *Main* method to create objects of Shape, Rectangle and Ellipse and call their Calculate… and ToString() methods. Use `Console.WriteLine` to show results on screen.
2. Build/Run the *ConsoleAppClientProject*, solve any unexpected issues and test the outcome. Also use step-by-step debugging to see the inner working of the application. The following table contains values that may be used:

Object	Length	Width	Perimeter	Area
Shape	42	13	-1	-1
Rectangle	2	3	10	6
Ellipse	10	6	± 25.91	± 47.12

Part 4: Creating and Using additional Shapes

Add two more Classes to the ShapesClassLibrary (Square & Circle)

1. In the ShapesClassLibrary create a **Square** class that inherits from Rectangle. These are the only two things that need to be added, thanks to the power of object orientation 😊
    a. add the following constructor:
    `public Square(double side) : base(side, side) { }`
    b. *override* the **ToString** method and make sure it returns a string telling something like 'I am a square with measures *so-and-so*…'
2. Add a second class as a subclass of Ellipse called **Circle**. Make sure your Circle class contains the following characteristics and code:
    a. it should inherit from class Ellipse
    b. one *public constructor* taking **diameter** as its only parameter, using *almost* the same code as used in step 1 a. of the previous step.



c. two public *override* methods called **CalculatePerimeter** and **CalculateArea** both returning a *double* but this time returning calculated circle perimeter ($2\pi r$) and area (πr^2) values respectively.

Prepare and Perform another Test Drive

3. In *ConsoleAppClientProject*, open file *Program.cs* and add code to the *Main* method to instantiate objects of the classes Square and Circle. Call their Calculate… and ToString methods and show their results on screen.
4. Build/Run the *ConsoleAppClientProject*, solve issues (if any) and analyse the results. Use step-by-step debugging in order to comprehend the way the application functions. The following table contains some test values:

Object	Side	Diameter	Perimeter	Area
Square	5	n/a	20	25
Circle	n/a	9	± 28.27	± 63.62

Part 5: Making sure No Objects can be Instantiated Directly from the Shape Class

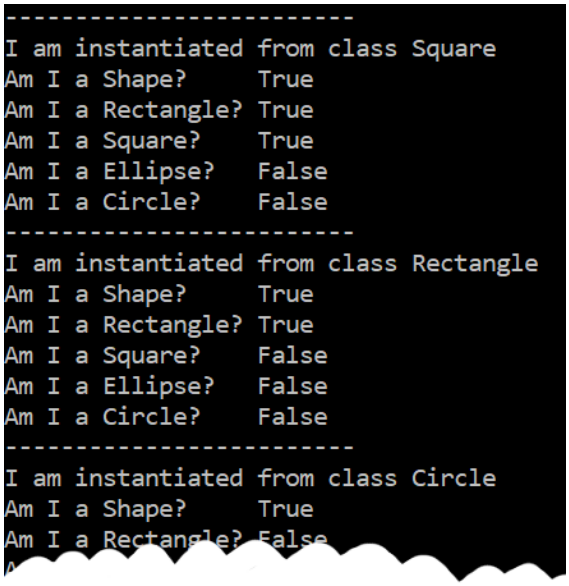
The Shape class actually serves the purpose of a prototype or base and is not meant to have a 'physical' representation in the 'real world' because its concept seems too abstract.

1. Add the keyword **abstract** to the Shape class and try to build/compile the solution.
2. If somewhere the code **new Shape(…)** is used you will see an error indicating an instance of an abstract class cannot be created. Remove or comment all code from the ConsoleAppClientProject that causes this error.
3. The *ToString* method in the *Shape* class is superfluous and can be removed/commented now.
4. Both *Calculate…* methods in the *Shape* class do not need their implementation. But they are important to make sure any derived class should contain an implementation of that method. This is done by replacing the keyword virtual by the keyword **abstract** to each of the *Calculate…* methods. Both method *bodies* need to be removed as well.

Part 6: Demonstrating some Polymorphic Behavior

Add some code in *Program.cs* in method *Main* after the creation of the different kinds of shapes made earlier, to perform the following:

1. Create a *generic* **List** of type/class **Shape**. Add all previously instantiated shape objects to this list.
2. Loop through this list and output the following per shape:
   a. the name of the type of the object
   b. for each shape in the collection, print "Am I a Shape?" True/False, "Am I a Rectangle? True/False, etc. (*hint*: use the **is** operator). The output should something look like this:



3. Save, run and test your work.

Part 7: Adding an extra Property to the Circle Class

1. Add a *read-only* property called **Radius** to the *Circle* class. Try to use the so-called *expression body* syntax to return half of the Length (or Width) value.
2. Inside the *loop* created in Part 6 step 2 add an *if*-statement that determines whether the shape object at hand is a Circle. If so, convert that shape object into a Circle object and read and display its *Radius* property, as follows: "I am a Circle with radius *so-and-so*".

Part 8: Adding a Color Enum and Properties to the Shape Class

1. In file Shape.cs, add a public enum at the *namespace level*. Call it **ShapeColor** and use these values: `Unknown, Red, Green, Blue, Yellow, Magenta, Cyan`
2. Add two properties and their belonging backing fields to the Shape class. Name these properties **BackgroundColor** and **BorderColor**. To insert a helpful piece of code, use the following *code snippet*: at the desired insert point *type* `propfull` and then press the [TAB] keyboard key twice.
3. In *Program.cs* in method *Main* set these new properties for a couple of instantiated shapes.
4. Inside the *loop* created in Part 6 step 2 read and display these color names for the shape at hand.

Part 9: Publishing, Subscribing to and Handling Events

Create the EventArgs Class to enable additional Data Transfer with the Event

1. Add a new Class file to the *ShapesClassLibrary* named **ColorChangedEventArgs.cs**
2. Make sure the class code looks as follows:

```
public class ColorChangedEventArgs : EventArgs
{
  public ShapeColor OldColor { get; set; }
  public ShapeColor NewColor { get; set; }
}
```

Declare the necessary Events and their belonging Methods to Raise these Events

1. At the top of class *Shape* add these two lines of code:

```
public event EventHandler<ColorChangedEventArgs> BackgroundColorChanged;
public event EventHandler<ColorChangedEventArgs> BorderColorChanged;
```

2. At the bottom of class *Shape* add these lines of code:

```
protected virtual void OnBackgroundColorChanged(ColorChangedEventArgs e) =>
  BackgroundColorChanged?.Invoke(this, e);
protected virtual void OnBorderColorChanged(ColorChangedEventArgs e) =>
  BorderColorChanged?.Invoke(this, e);
```

Adjust both Color Setter Properties to Call the Raise Event Methods

1. Change the *set* part of both properties **BackgroundColor** and **BorderColor** to check whether the incoming **value** different from the one already stored in its belonging backing field.
2. In case of different values, store the current color value in a separate variable and store the incoming **value** in its belonging backing field.
3. Create an object of type **ColorChangedEventArgs** and fill both properties with the appropriate values.
4. Call the belonging *On…ColorChanged* method, passing it the *ColorChangedEventArgs* object just created.

Subscribe to the Events

1. In *ConsoleAppClientProject*, open file *Program.cs* and add the yellow marked line of code to the *Main* method *after* instantiation of some (or all) of the shape objects, but *before* changing their colors:

```
var myFirstRectangle = new Rectangle(2,3);

myFirstRectangle.BackgroundColorChanged +=

myFirstRectangle.BackgroundColor = ShapeColor.Blue;
myFirstRectangle.BorderColor = ShapeColor.Red;
```

As indicated, press the [TAB] keyboard key to automatically generate the belonging event handler method.

You could also *change/rename* the method name during this step to something less specific, like *Shape_BackgroundColorChanged* so its name sounds more suitable to be reused by other shapes...

2. Perform the same step for the *BorderColorChanged* event
3. At the bottom of the file two event handler methods have been added now. Make sure the output of those methods is something like this:
```
Console.WriteLine($"The border color of a {sender.GetType().Name} has changed from {e.OldColor}
  to {e.NewColor}.");
```
4. Build/Run the *Solution*, solve any unexpected issues and check whether the event handler texts created before are shown in the output.

Part 10: Enable the Shapes to be Drawn

Add additional References to be able to Draw WPF Shapes

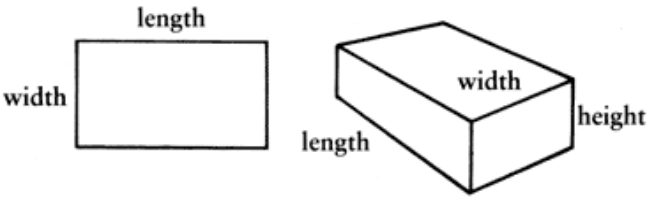
6. In the Solution Explorer select *ShapesClassLibrary*.
7. In the Visual Studio top menu select Project > Add Reference…
8. Open Assemblies > Framework and check these libraries and click [OK]:
   - ☑ WindowsBase
   - ☑ PresentationCore
   - ☑ PresentationFramework

Add two Draw Related Methods to the Shape Class

1. In order to make a clear difference between all *our* Shape classes and the ones Microsoft created for WPF, at the top of file Shape.cs add the following *alias*:
```
using WPF = System.Windows.Shapes;
```
2. Because WPF has no notion of the color Unknown, remove `Unknown` from the ShapeColor enum and add these two: `Black, White`
3. At the end of the Shape class, add the following method:
```
public abstract WPF.Shape Draw();
```
4. Below that abstract method, add this method that sets several *WPF Shape* properties in order to draw *2D* shapes:
```
protected void PrepareDrawingGroundwork(WPF.Shape shapeToBeDrawn)
{
  if (shapeToBeDrawn != null)
  {
    shapeToBeDrawn.Width = Length;
    shapeToBeDrawn.Height = Width;

    var fillColor = (Color)ColorConverter.ConvertFromString(BackgroundColor.ToString());
    var strokeColor = (Color)ColorConverter.ConvertFromString(BorderColor.ToString());

    shapeToBeDrawn.Fill = new SolidColorBrush(fillColor);
    shapeToBeDrawn.Stroke = new SolidColorBrush(strokeColor);

    shapeToBeDrawn.StrokeThickness = 5;
  }
}
```

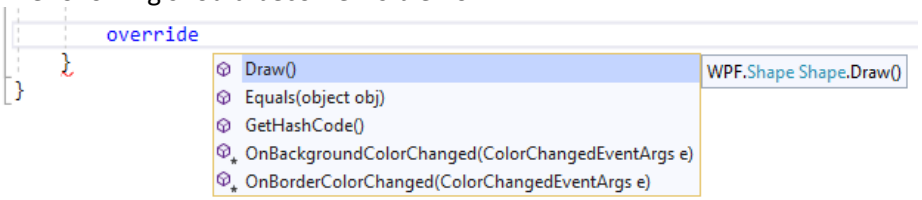
Use the Light Bulb in Visual Studio to fix the *does not exist/could not be found* errors…

The reason for the shapes to be drawn on the computer screen to use *Width* for paper drawn *Length* and *Height* for paper drawn *Width* can be explained using this drawing. The left image is how you would draw a shape on paper, looking from *above*, using *length* and *width*. Computer screens are considered to be looked upon from the *side*. So looking at the right figure from the right, *width* and *height* are used to define and measure computer screen shapes (instead of *length* and *width*).



Override the Draw Method in the Rectangle and Ellipse Classes

1. Add this *using* statement at the top of file *Rectangle.cs*:
   ```
   using WPF = System.Windows.Shapes;
   ```
2. In file Rectangle.cs inside the Rectangle class type the following:
   `override` directly followed by a [SPACE]
   The following should become visible now:

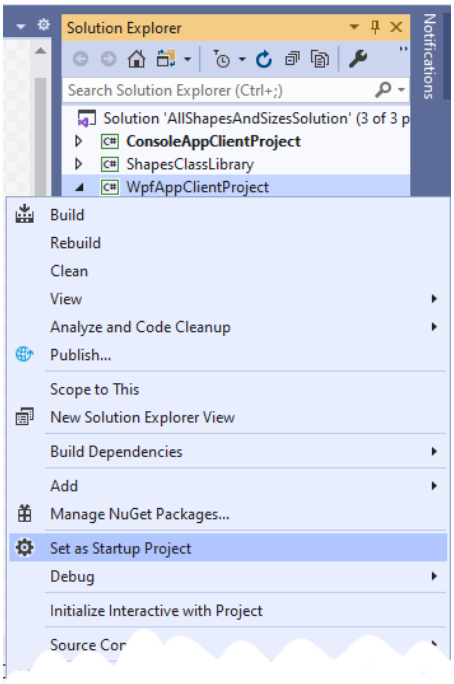


   In this case select `Draw()` by pressing [ENTER] or [TAB] or *double click* with the *left mouse button*.
3. *Remove* the generated line that throws a `NonImplementedException`.
4. Make sure that for the Rectangle class the complete method should as follows:
   ```
   public override WPF.Shape Draw()
   {
     var rectangleToBeDrawn = new WPF.Rectangle();

     PrepareDrawingGroundwork(rectangleToBeDrawn);

     return rectangleToBeDrawn;
   }
   ```
5. Perform steps 1, 2, 3 and 4 for the *Ellipse.cs* file as well. NOTE: Inside method `Draw()` a `new WPF.Ellipse` should be instantiated and the variable name should then be `ellipseToBeDrawn`.

Because the WPF Shape class (`namespace System.Windows.Shapes`) is defined as `abstract` (just as our own Shape class), we need to create an object of the concrete classes WPF.Rectangle or WPF.Ellipse in our derived classes and pass that instance to base class method `PrepareDrawingGroundwork` in order set several common/general drawing properties.

Part 11: Using Windows Presentation Foundation (WPF) to Display the Shapes

Add a WPF Project to the current Solution

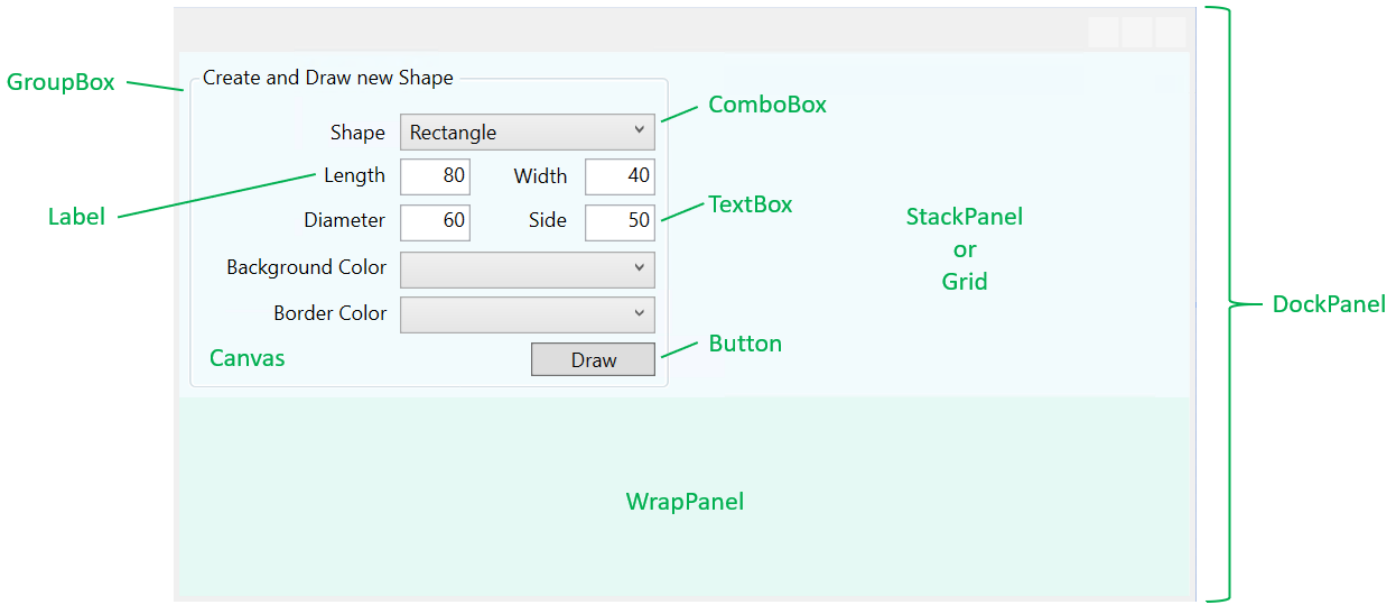


1. In the Visual Studio top menu select File > Add > New Project…
2. In the now appearing dialog entitled "Add a new project" use the search text box and type *WPF*.
3. In the shown results select "WPF App (.NET Framework).
4. Enter for **Project name**: *WpfAppClientProject* and click [Create]
5. In the Solution Explorer panel *right-click* the project file *WpfAppClientProject* that was just created and select *Set as Startup Project* as shown in the picture.
6. Add a *reference* from this project to the *ShapesClassLibrary* so our Shapes classes can be used.

Create the User Interface using the Designer and XAML

1. From the Solution Explorer open file *MainWindow.xaml* and create a user interface similar to the one in the picture below, using the indicated control types.
2. Give the controls meaningful names using their `x:Name` attribute by following the guideline to postfix that name with its complete control type name, like `diameterTextBox`, `backgroundColorComboBox` or `drawButton`.
3. The *ComboBox* containing the Shapes to choose from can be filled using XAML like this:

```
<ComboBox x:Name="shapeComboBox" ... >
  <ComboBoxItem>Circle</ComboBoxItem>
  <ComboBoxItem>Ellipse</ComboBoxItem>
  <ComboBoxItem IsSelected="True">Rectangle</ComboBoxItem>
  <ComboBoxItem>Square</ComboBoxItem>
</ComboBox>
```



Add C# Code behind the XAML Window

1. From the Solution Explorer open file *MainWindow.xaml.cs* and make sure the following using statements are among the ones already present:

```
using ShapesClassLibrary;
using WPF = System.Windows.Shapes;
```

2. Add the following variable inside the `MainWindow` class:

```
// declare a (still empty) variable for the shape to be drawn
private Shape latestShape = null;
```

3. In constructor method `public MainWindow` after calling `InitializeComponent` add these lines:

```
// fill the ComboBoxes with the possible colors
backgroundColorComboBox.ItemsSource = Enum.GetValues(typeof(ShapeColor));
borderColorComboBox.ItemsSource = Enum.GetValues(typeof(ShapeColor));

// set ComboBoxes to their default values
backgroundColorComboBox.SelectedItem = ShapeColor.White;
borderColorComboBox.SelectedItem = ShapeColor.Black;
```

Respond to the Button Click Event

1. Open the *MainWindows.xaml* file and *double-click* the [Draw] button in the Designer part of the screen. Visual Studio should now jump to the belonging *C#* code file named *MainWindow.xaml.cs* and this *event handling* code should have been *generated*:

```
private void drawButton_Click(object sender, RoutedEventArgs e)
{

}
```

2. Add code to this method to validate that the values entered in the TextBoxes are integer values and greater than zero. You can use `int.TryParse` for this challenge. Use `MessageBox.Show` to display user errors.
3. Add a switch statement to instantiate the selected Shape subclass. Two hints:
   - Use the `Text` property of the shapeComboBox to get the selected value.
   - Use class level variable `latestShape` created in step 2 of the previous section to store the instantiated Shape subclass into.
4. Use this code to set both color properties of the shape according to the ComboBoxes:

```
latestShape.BackgroundColor = (ShapeColor)Enum.Parse(typeof(ShapeColor),
  backgroundColorComboBox.SelectedItem.ToString());
latestShape.BorderColor = (ShapeColor)Enum.Parse(typeof(ShapeColor),
  borderColorComboBox.SelectedItem.ToString());
```

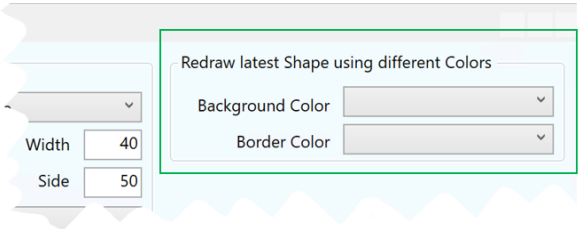
5. To finally be able to draw 2D shapes and display them in a panel, add these lines at the end of the method:

```
// actually draw the shape
var drawnShape = latestShape.Draw();

// add some space around it
drawnShape.Margin = new Thickness(5);

// add the drawn shape to the panel
myWrapPanel.Children.Add(drawnShape);

// sync the 'redraw' ComboBoxes with the shape just drawn
latestShapeBackgroundColorComboBox.SelectedItem = latestShape.BackgroundColor;
latestShapeBorderColorComboBox.SelectedItem = latestShape.BorderColor;
```

6. Build/Run the *Solution*, solve any unexpected issues and try to show several shapes in different colors.

Part 12: Adjusting the Colors of Drawn Shapes

Expand the User Interface



1. Use the area indicated by the green frame in the picture to expand the current user interface.
2. Name the controls in a meaningful way postfixing their functional purpose with the complete control type name concerned, as explained before.
3. Make sure to fill both ComboBoxes with our possible Shape class colors the same way the other two ComboBoxes are initialized.

Add a method to Redraw the Shape

1. In file MainWindow.xaml.cs, inside class MainWindow, add a *private void* method named **RedrawShape**.
2. Inside that method, perform the following steps:
   a) remove the last added child (if present) from the *WrapPanel*
   b) redraw the shape

c) add some space around it

d) finally add the redrawn shape to the WrapPanel

Subscribe to both our Shape Object ColorChanged Events

- In file MainWindow.xaml.cs, inside the `drawButton_Click` method, directly before *actually drawing the shape* add *one* event handler method to handle *both* ColorChanged events that are raised from our *latest Shape* object. Make sure the *RedrawShape* method is called when *either* event is raised.

Events can be handled using a *separate* event handler *method*. But if the code to be executed is not called from other places, a so-called *inline event handler* could also be used to attach an *anonymous* method to the event:

```
latestShape.BackgroundColorChanged += delegate (object o, ColorChangedEventArgs args)
{
    RedrawShape();
};
```

When a lambda expression is used, the syntax is even shorter:

```
latestShape.BorderColorChanged += (o, args) => RedrawShape();
```

see also https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-functions

Subscribe to both SelectionChanged Events of the Redraw ComboBoxes

1. Add an event handler *method* to each of the *redraw* Color ComboBoxes and change the belonging **latestShape** object Color property accordingly. Two hints:
    - check whether *latestShape* has already been *instantiated* first
    - use the `Enum.Parse` method mentioned earlier to set the correct color

Part 13: Enable and Disable Controls when Necessary (Bonus, If Time Permits)

1. Disable or enable TextBoxes that are (ir)relevant due to the current Shape choice to be drawn. For example when selecting a circle, only the *Diameter* TextBox needs to be accessible. Hint: use property `IsEnabled`.
2. Also define a 'starting' position; which TextBox(es) should be enabled/disabled when the application starts up.
3. Right after starting the application (when no Shape has been drawn yet) the 'redraw' ComboBoxes should be disabled. As soon as one shape is drawn, both are to be enabled and should stay that way.