

Module 10 – Inheritance and polymorphism.

This module covers two advance topics in object-oriented programming: inheritance and polymorphism. To continue with the rest of the chapter, please download the zipped Module10_student project file in OdtuClass and open it in Visual Studio.

1) Inheritance

In object-oriented programming, **Inheritance** means deriving a new class (called *derived* class) from an existing class (called *base* class). Typically, the base class represents a rather generic object (such as shape), whereas the derived class (such as rectangle) is a more specialized version of the generic object. Inheritance relies on “**is a relationship**” between the derived and base classes. Some examples are:

- A flower is a plant.
- A golden-retriever is a dog.
- A triangle is a shape.

A derived class can have additional properties and methods in addition to those inherited from the base class. For example, a shape class can have LocationX, LocationY, and Color properties that are common to all shape objects. A rectangle class derived from the shape class can have properties such as width and height (that are distinct to a rectangle), in addition to those owned by the base class (e.g., LocationX, LocationY, and Color).

We will work on an example to understand how inheritance works in practice. Imagine that in our application we need to create different types of shapes: triangle, rectangle, and circle. All these shapes share four common properties: an int property named **Id**, a string property called **Name**, a string property called **BorderColor**, and a string property named **FillColor**.

Circle class will have three properties in addition to the shared properties: an **int** property named **Diameter**, an **int** property named **Angle**, and a **double** property named **Radius**. Create a folder named **Model** in your project, and inside this folder, create the Circle class file. The definition of the **Circle** class is provided below.

```
class Circle
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string BorderColor { get; set; }
    public string FillColor { get; set; }

    public int Diameter{ get; set; }
    public int Angle{ get; set; }
    public double Radius { get { return Diameter / 2; } }
}
```

`Triangle` class will have four additional properties: an `int` property named **Height**, an `int` property **Base**, an `int` property **SideLeft**, an `int` property **SideRight**. The definition of the `Triangle` class is provided below. Add this class to the Model folder.

```
class Triangle
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string BorderColor { get; set; }
    public string FillColor { get; set; }

    public int Height { get; set; }
    public int Base { get; set; }
    public int SideLeft { get; set; }
    public int SideRight { get; set; }
}
```

`Rectangle` class will have four additional properties: an `int` property named **Width**, and an `int` property **Length**. The definition of the `Rectangle` class is provided below. Add this class to the Model folder.

```
class Rectangle
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string BorderColor { get; set; }
    public string FillColor { get; set; }

    public int Width { get; set; }
    public int Length { get; set; }
}
```

You might have already realized that this is a rather inefficient approach. All three classes share four common properties, and we repeat them in each class. Not only this will result in duplicated code but will also make it harder to change the common properties or add a new common property.

A more efficient approach would be to define a base class called `Shape` which will hold all common properties. Definition of the `Shape` class is provided below. Add this class to the Model folder.

```
class Shape
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string BorderColor { get; set; }
    public string FillColor { get; set; }
    public string DisplayText { get { return Id.ToString() + ". " + Name; } }
}
```

Now, we will revisit the `Triangle`, `Circle`, and `Rectangle` classes. These classes will derive from the `Shape` class (i.e., base class) to **inherit** the properties shared by all shape objects. Below is the updated definition of the `Triangle` class. Please pay attention to the change in the class header. `Triangle : Shape`. This format is used to indicate that `Triangle` derives from the `Shape` class.

```

class Triangle : Shape
{
    public int Height { get; set; }
    public int Base { get; set; }
    public int SideLeft { get; set; }
    public int SideRight { get; set; }
}

```

We will also update the `Circle` and `Rectangle` classes so that they also derive from the `Shape` class. In the updated class definitions, the common properties should be removed.

```

class Circle : Shape
{
    public int Diameter{ get; set; }
    public int Angle{ get; set; }
    public double Radius { get { return Diameter / 2; } }
}

class Rectangle : Shape
{
    public int Width { get; set; }
    public int Length { get; set; }
}

```

2) Inheritance in practice

We will put these ideas in practice by working on an application. You should download the project file from OdtuClass. The interface of the application (see Figure 1) is already created as shown below and some minimal code is included. This application will allow users to create different shapes, and to list the shapes created in a list box (available in the Shape List tab page).

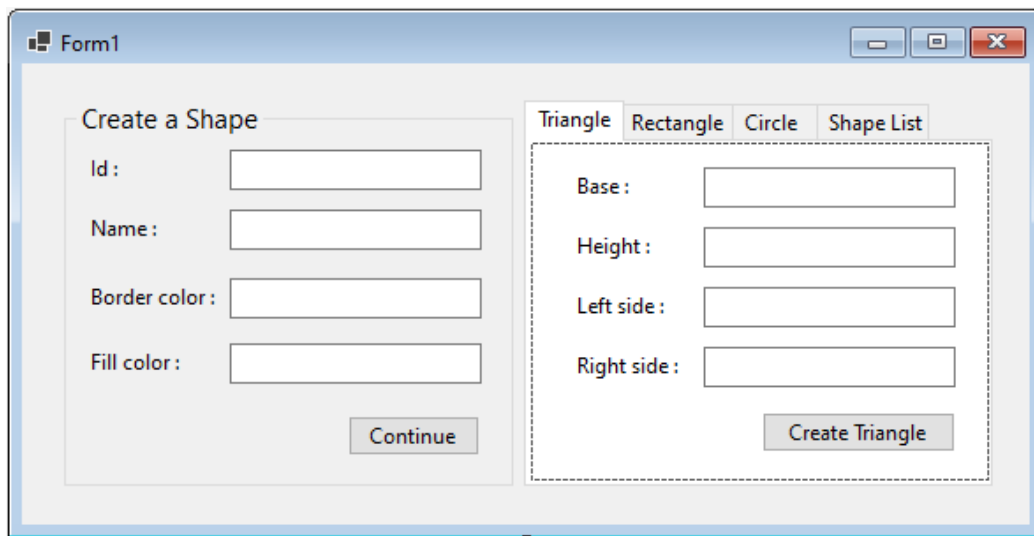


Figure 1. The main interface of the application.

When the application is run, the form will be displayed in a smaller window as shown in Figure 2. When the Continue button is clicked, the form will be extended to the right to show the options for creating the type of shape to be created (see Figure 3). This functionality is already programmed.

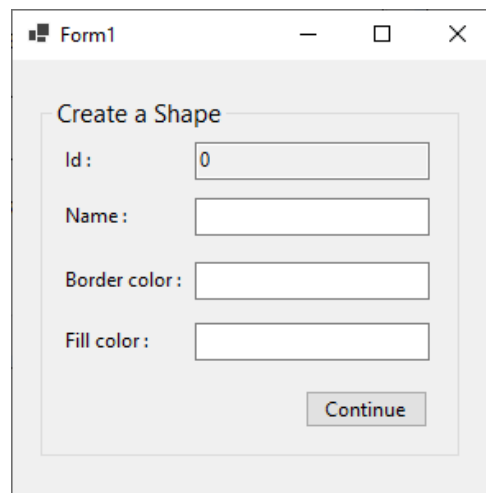


Figure 2. The first screen when the application is run.

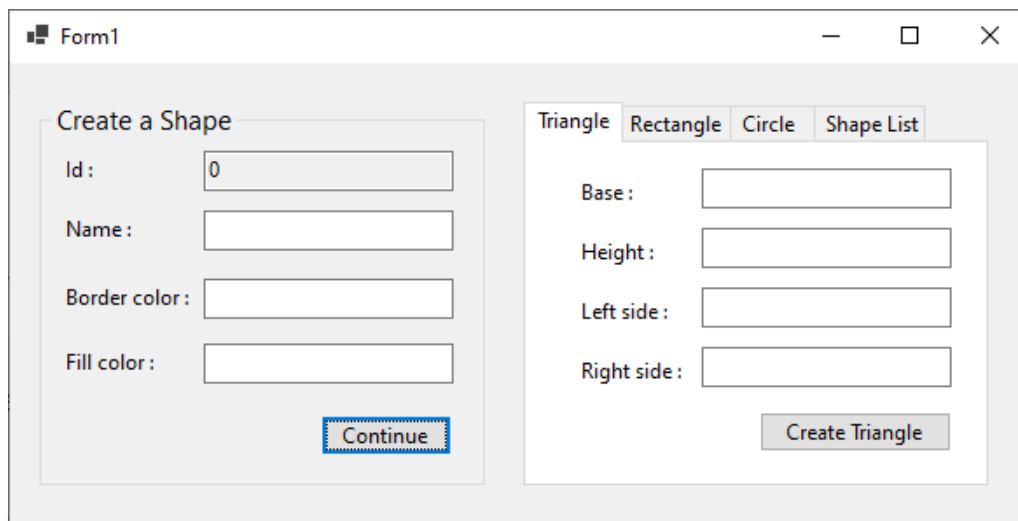


Figure 3. The screen extended after Continue is clicked.

When the **Create Triangle** button is clicked, a new **Triangle** object should be created (based on the user input) and shown in the list box inside the Shape List tab page. To implement this, please double click on the Create Triangle button to create its click event handler.

Inside the click event handler, please write the following code to create an empty **Triangle** object and set its property values based on the data provided by the user.

```
//Create an empty Triangle object named myTriangle
Triangle myTriangle = new Triangle();

//Set values for the common properties
myTriangle.Id = controlId;
controlId = controlId + 1;
```

```

myTriangle.Name = txt_name.Text;
myTriangle.BorderColor = txt_bordercolor.Text;
myTriangle.FillColor = txt_fillcolor.Text;

//Set values for the distinct properties
myTriangle.Base = int.Parse(txt_triangleBase.Text);
myTriangle.SideLeft = int.Parse(txt_leftSide.Text);
myTriangle.SideRight = int.Parse(txt_rightSide.Text);
myTriangle.Height = int.Parse(txt_triangleHeight.Text);

```

After the object is properly updated with user data, you can use the following code to add this object to `triangles`, which is a `BindingList` type variable defined at class level to hold multiple `Triangle` type objects. `triangles` will be updated each time a new triangle object is created.

```

//Add the myTriangle object to a list of triangles
triangles.Add(myTriangle);

```

Next, the following code will can bind `triangles` to `lst_shapes` to list the `Triangle` objects created. The `DisplayMember` property of `lst_shapes` is set to "DisplayText", which is a property of the `Shape` class that returns the Id and Name properties combined.

```

//Display the list of triangles
lst_shapes.DataSource = triangles;
lst_shapes.DisplayMember = "DisplayText";
lst_shapes.ValueMember = "Id";

```

Last, what goes inside the click event handler of Create Triangle is the following code. This code will switch the current tab to Shape List tab where the new objects created are listed. Also, `ResetForm` method will be called to reset all the fields to empty.

```

//Switch to the Shape List tab
tab_shapes.SelectedTab = tab_shapeList;

//Reset the form
ResetForm();

```

You can run and test the application. Please enter some data as shown in Figure 4.

The screenshot shows a Windows application window titled "Form1". It contains two main panels. The left panel, titled "Create a Shape", has four text boxes: "Id:" with the value "0", "Name:" with "Big Triangle", "Border color:" with "Red", and "Fill color:" with "White". Below these is a "Continue" button. The right panel, titled "Triangle", has four text boxes: "Base:" with "12", "Height:" with "8", "Left side:" with "5", and "Right side:" with "5". Below these is a "Create Triangle" button. At the top of the right panel are four tabs: "Triangle" (selected), "Rectangle", "Circle", and "Shape List".

Figure 4. Creating a triangle object.

Next, click on the Create Triangle button. You should see the new triangle object being listed in the Shape List tab, as shown in Figure 5.

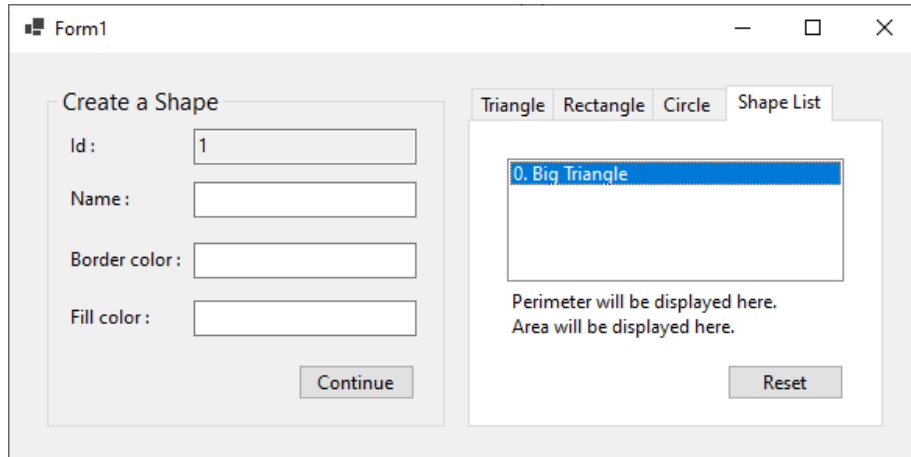


Figure 5. The new triangle object is listed.

We will continue building our application by implementing the **Create Rectangle** button's click event handler (see Figure 6). Please double click on the Create Rectangle button to create its click event handler.

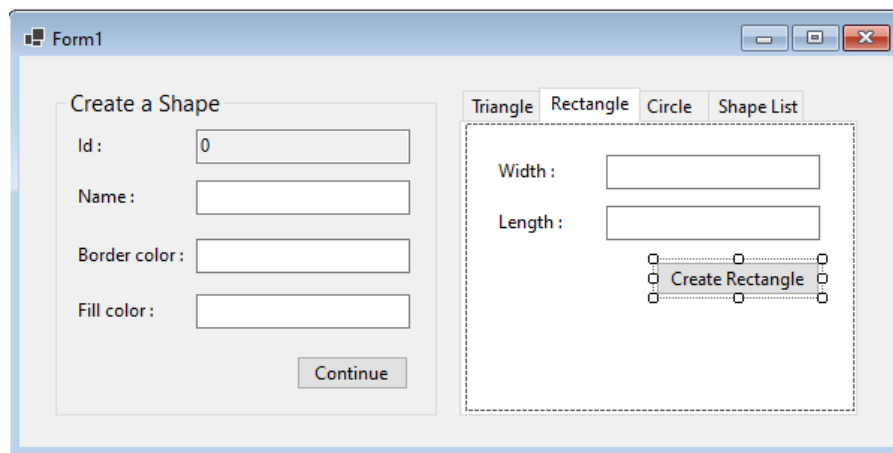


Figure 6. Clicking on the Create Rectangle button.

Click event handler of the Create Rectangle button should have a very similar code to what we have already for the previous button. The main difference would be to create a **Rectangle** object instead of **Triangle**. At this point we can take advantage of **Inheritance** to somehow avoid writing repetitive code.

To begin with, when creating a new shape (no matter it is a rectangle, triangle, or a circle), it will have four common properties: Id, Name, BorderColor, and FillColor. All these property values will be set based on user data. We can define a method that sets the values of these properties for any class object that is derived from the **Shape** base class.

Figure 7 shows the definition of the SetSharedProperties method which accepts any **Shape** object as the parameter. In this case, **Triangle**, **Rectangle**, and **Circle** objects can be passed as the parameter since each of them is a shape (as they derive from the **Shape** class).

```

private void SetSharedProperties(Shape shape)
{
    shape.Id = controlId;
    controlId = controlId + 1;
    shape.Name = txt_name.Text;
    shape.BorderColor = txt_bordercolor.Text;
    shape.FillColor = txt_fillcolor.Text;
}

```

Figure 7. Defining the SetSharedProperties method.

Now, we can call this method inside the click event handler of Create Rectangle button. In the code shown in Figure 8, first a new `Rectangle` object is created, named `myRectangle`. Then, `SetSharedProperties` is called to set the values for the common properties inherited from the `Shape` class. Next, the values of the `Width` and `Length` properties are set properly based on user input.

```

private void btn_createRectangle_Click(object sender, EventArgs e)
{
    Model.Rectangle myRectangle = new Model.Rectangle();

    //Set values for the common properties
    SetSharedProperties(myRectangle);

    //Set values for the distinct properties
    myRectangle.Width = int.Parse(txt_width.Text);
    myRectangle.Length = int.Parse(txt_length.Text);
}

```

Figure 8. Calling the SetSharedProperties method inside the click event handler.

Now that the object is created and initialized properly, we can display it in the listbox control. As you may remember, for the `Triangle` object, we used a `BindingList` that can hold only `Triangle` type objects. However, this is not sufficient because we want to store multiple types of objects in a list and display them together in a listbox.

We can apply a more effective approach thanks to Inheritance. We will create a `BindingList` that can hold `Shape` type objects. Since `triangle`, `circle`, and `rectangle` are also a `shape`, they can be stored in this list. Let's go ahead and create the `BindingList` variable, named `shapes`, to hold `Shape` type objects. Please note that this variable should be defined at class level as shown in Figure 9.

```

public Form1()
{
    InitializeComponent();
}

int controlId = 0;
BindingList<Shape> shapes = new BindingList<Shape>();

```

Figure 9. Defining the shapes variable.

Next, we will add the `myTriangle` object to the `shapes` list, then we will bind the `shapes` list to `lst_shapes`, as shown below.

```

//Add the myRectangle object to a list of triangles
shapes.Add(myRectangle);

//Display the list of triangles
lst_shapes.DataSource = shapes;
lst_shapes.DisplayMember = "DisplayText";
lst_shapes.ValueMember = "Id";

```

We need to bind shapes to lst_shapes each time a new `Triangle`, `Circle`, or `Rectangle` objects are created. We can eliminate this repetition by creating a method that binds shapes list to lst_shapes. Then, we need to call this method only once when the page is loaded. Since shapes is a **BindingList** type, any changes in shapes will be automatically reflected to lst_shapes, where the shapes list is bounded as the data source. The complete code is shown below.

```

private void BindData()
{
    lst_shapes.DataSource = shapes;
    lst_shapes.DisplayMember = "DisplayText";
    lst_shapes.ValueMember = "Id";
}

private void Form1_Load(object sender, EventArgs e)
{
    Width = 310;
    BindData();
}

```

Figure 10. Binding the data.

After these changes, the final code in the click event handler of btn_createRectangle button should look like this:

```

private void btn_createRectangle_Click(object sender, EventArgs e)
{
    Model.Rectangle myRectangle = new Model.Rectangle();

    //Set values for the common properties
    SetSharedProperties(myRectangle);

    //Set values for the distinct properties
    myRectangle.Width = int.Parse(txt_width.Text);
    myRectangle.Length = int.Parse(txt_length.Text);

    //Add the myRectangle object to a list of triangles
    shapes.Add(myRectangle);

    //Switch to the Shape List tab
    tab_shapes.SelectedTab = tab_shapeList;

    //Reset the form
    ResetForm();
}

```

Figure 11. Binding the data.

You should go back and revise the `btn_createTriangle_Click` click event handler. With the addition of some new methods, it should be simplified.

Following the same approach, also implement the click event handler for **Create Circle** (see Figure 12).

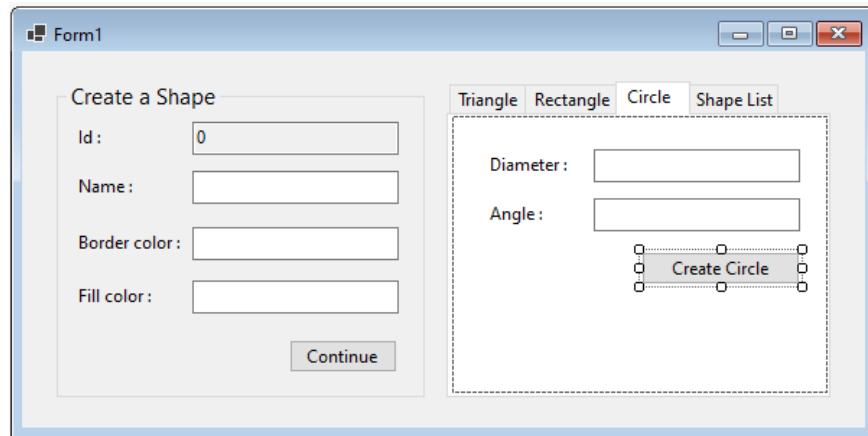


Figure 12. Creating click event handler for the Create Circle button.

After completing the click event handlers for buttons, please run your application and add three different shapes. The list box should be able to display all different types of shapes as shown in Figure 13 below.

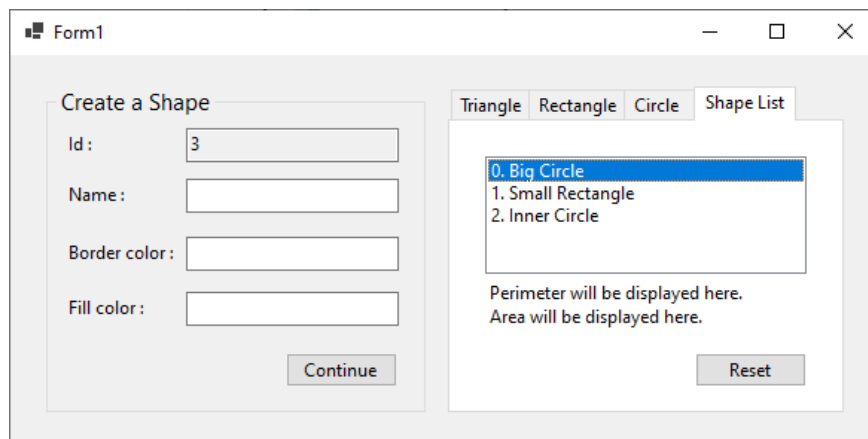


Figure 13. Different shapes are added and displayed in the list box.

3) Polymorphism

The term **Polymorphism** means that objects can exist in different forms. In terms of object-oriented programming, Polymorphism allows derived classes to have different implementation for the methods inherited from the base class. Thanks to the polymorphism, program can identify which method to call depending on the object type. We will apply polymorphism in our current example.

We will add two new methods to the `Shape` class: `CalculatePerimeter` and `CalculateArea`. These methods will return 0, since the length, width, or diameter of the shape is not known yet. `CalculatePerimeter` and `CalculateArea` methods will have the `virtual` keyword in the header, as seen below. Methods declared with `virtual` keyword are allowed to be overridden by the derived class.

```

class Shape
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string BorderColor { get; set; }
    public string FillColor { get; set; }

    public string DisplayText { get { return Id.ToString() + ". " + Name; } }

    public virtual double CalculatePerimeter()
    {
        return 0;
    }

    public virtual double CalculateArea()
    {
        return 0;
    }
}

```

Next, we will update the definition of the `Rectangle` class. As shown below, the `Triangle` class overrides the `CalculatePerimeter` and `CalculateArea` methods, and provides a new definition for these methods based on how a triangle's perimeter and area should be calculated. Pay attention to the `override` keyword in the method header. The `override` indicates that the method in the derived class overrides a method in the base class.

```

class Triangle: Shape
{
    public int Height { get; set; }
    public int Base { get; set; }
    public int SideLeft { get; set; }
    public int SideRight { get; set; }

    public override double CalculatePerimeter()
    {
        return Base + SideLeft + SideRight;
    }

    public override double CalculateArea()
    {
        return Base * Height / 2;
    }
}

```

We will define a new `Triangle` object, called `myTriangle`, whose `Base` is 10 and `Height` is 15.

```

Triangle myTriangle = new Triangle();
myTriangle.Base = 10;
myTriangle.Height = 15;

```

We will call the `CalculateArea` method to calculate the area for the `myTriangle` object. Then we will print the computed area using `MessageBox.Show` method.

```

double area = myTriangle.CalculateArea();

```

```
MessageBox.Show("Area of the triangle is " + area.ToString());
```

The following popup window should appear printing the area of the triangle correctly.

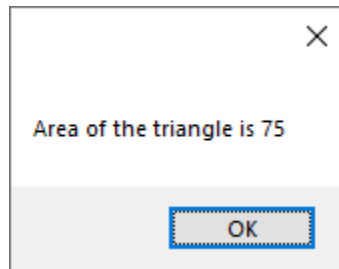


Figure 14. Printing the area of the triangle.

We will update the `Circle` and `Rectangle` classes accordingly by overriding the `CalculatePerimeter` and `CalculateArea` methods.

```
class Circle: Shape
{
    public int Diameter{ get; set; }
    public int Angle{ get; set; }
    public double Radius { get { return Diameter / 2; } }

    public override double CalculatePerimeter()
    {
        return Math.PI * Radius * 2;
    }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius * Angle /360;;
    }
}

class Rectangle: Shape
{
    public int Width { get; set; }
    public int Length { get; set; }

    public override double CalculatePerimeter()
    {
        return 2 * (Width + Length);
    }

    public override double CalculateArea()
    {
        return Width * Length;
    }
}
```

We are allowed to override properties as well. As an additional exercise, you can define the **DisplayText** property with `virtual` keyword, and override this property in the derived classes to set a different display text for different shapes.

4) Is a relationship in Polymorphism

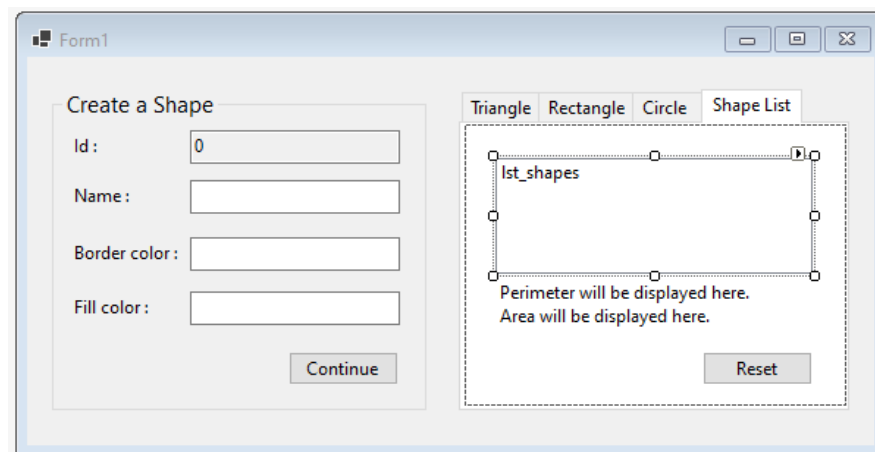
You may remember that there is a “Is a” relationship between the derived and bases classes. For example, a `Triangle` is not only a `Triangle` but is also a `Shape`. Thanks to this relationship, we can use a `Shape` class variable to reference a `Triangle` object. For example, look at the following code:

```
Shape myShape = new Triangle() { Base = 10, Height = 15 };  
double area = myShape.CalculateArea();  
  
MessageBox.Show("Area of the triangle is " + area.ToString());
```

In the code above, a `Triangle` object is created and assigned to a `Shape` variable called `myShape`. This assignment is legal because a `Triangle` object is also a `Shape` object. The `CalculateArea` method will compute the area of the triangle correctly because `myShape` will know that it is referencing a `Triangle` object.

5) Polymorphism in practice

We will apply polymorphism in our project to compute the perimeter and area of the item selected in the list box no matter if it is a triangle, rectangle, or circle. As shown in Figure 15, please double click on the `lst_shapes` to create its **SelectedIndexChanged** event handler.



```
private void lst_shapes_SelectedIndexChanged(object sender, EventArgs e)  
{  
    .....  
}
```

Figure 15. Creating the SelectedIndexChanged event handler.

This event handler will be executed when the selected item in `lst_shapes` is changed. We will write our code to compute the perimeter and area of the selected shape inside this event handler. Thanks to polymorphism this will be quite easy.

First, we obtain the **Id** of the selected shape by using **SelectedValue** property. We use the **Id** value inside the **Single** method to find the target shape inside the `shapes` list. The rest is quite straightforward. We

call the CalculatePerimeter and CalculateArea methods for the shape and print them in the labels properly. See the complete code in Figure 16.

```
private void lst_shapes_SelectedIndexChanged(object sender, EventArgs e)
{
    int shapeId = (int)lst_shapes.SelectedValue;
    Shape selectedShape = shapes.Single(s => s.Id == shapeId);

    double perimeter = Math.Round(selectedShape.CalculatePerimeter(), 2);
    lbl_perimeter.Text = "Perimeter : " + perimeter.ToString();

    double area = Math.Round(selectedShape.CalculateArea(), 2);
    lbl_shapeArea.Text = "Area : " + area.ToString();
}
```

Figure 16. Implementing the SelectedIndexChanged event handler.

Please note that, thanks to Inheritance and Polymorphism, the program knows the specific class of each shape selected (i.e., whether it is a triangle, rectangle, or circle) and calls the correct version of the CalculatePerimeter and CalculateArea methods depending on the type of the shape.

Please run and test your application. Define different types of shapes and check if the Perimeter and Area labels are properly updated as these shapes are selected in the list box. Some examples are shown below.

