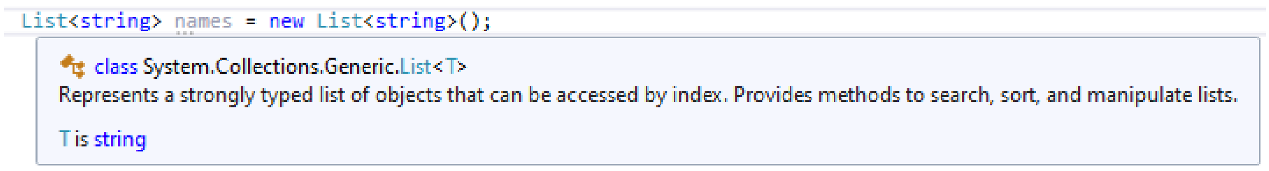
Module 9 – Introduction to the Classes and Objects.

C# is an object-oriented programming languages. That means, coding in C# involves creating many **objects** and using their *properties* and *methods* to perform some operations. For example, all controls (e.g., label, textbox, button, etc.) that we have used to design forms are actually objects. Lists are also another great example of the objects we used so far. To be able to create and use objects of a particular type, a `class` must be created. Below is the definition of a list object called `names`. If you mouse over the `List` keyword, a tooltip window will appear, where you can see that `List` is a `class`.



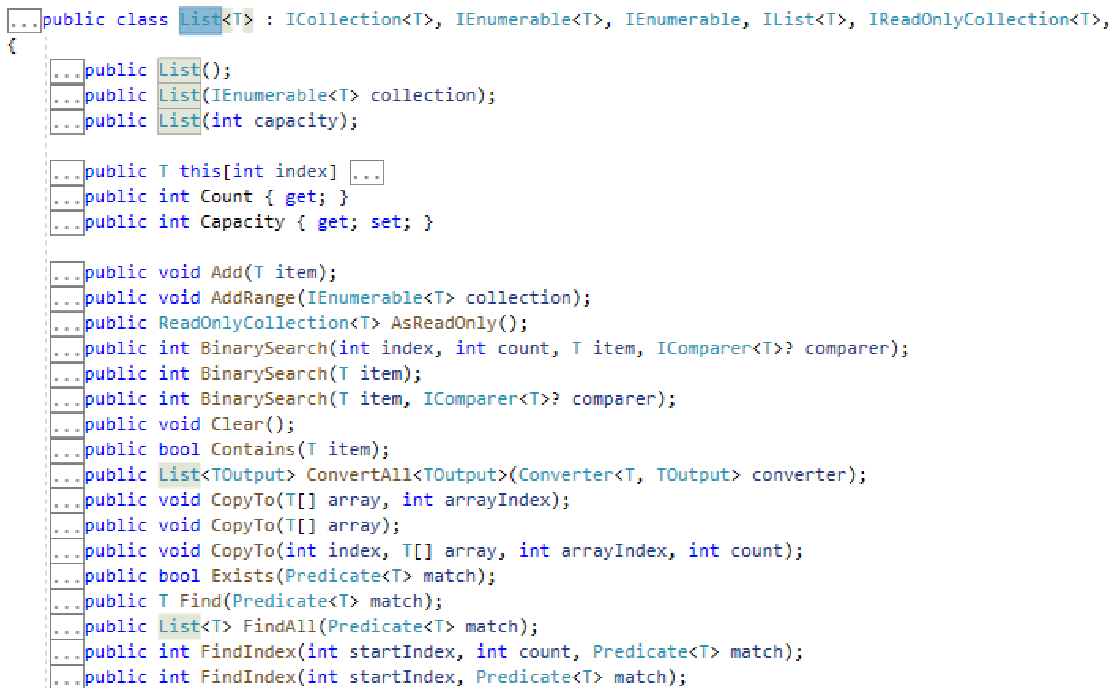
```
List<string> names = new List<string>();
```

class System.Collections.Generic.List<T>
Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

T is string

Figure 1. Defining names list object.

While the cursor is inside the List keyword, press F12 to view the definition of the List class. Its definition is quite long, and Figure 2 shows only part of it. You may remember that we used some **properties** (e.g., *Count*) and **methods** (*Add, Clear, Contains*) for List objects. They are all created in the definition of the `List class` as you can see in Figure 2. Any object created from `List class`, will have the same set of properties and methods as defined in the class declaration.



```
...public class List<T> : ICollection<T>, IEnumerable<T>, IEnumerable, IList<T>, IReadOnlyCollection<T>,
{
    ...public List();
    ...public List(IEnumerable<T> collection);
    ...public List(int capacity);

    ...public T this[int index] ...
    ...public int Count { get; }
    ...public int Capacity { get; set; }

    ...public void Add(T item);
    ...public void AddRange(IEnumerable<T> collection);
    ...public ReadOnlyCollection<T> AsReadOnly();
    ...public int BinarySearch(int index, int count, T item, IComparer<T>? comparer);
    ...public int BinarySearch(T item);
    ...public int BinarySearch(T item, IComparer<T>? comparer);
    ...public void Clear();
    ...public bool Contains(T item);
    ...public List<TOutput> ConvertAll<TOutput>(Converter<T, TOutput> converter);
    ...public void CopyTo(T[] array, int arrayIndex);
    ...public void CopyTo(T[] array);
    ...public void CopyTo(int index, T[] array, int arrayIndex, int count);
    ...public bool Exists(Predicate<T> match);
    ...public T Find(Predicate<T> match);
    ...public List<T> FindAll(Predicate<T> match);
    ...public int FindIndex(int startIndex, int count, Predicate<T> match);
    ...public int FindIndex(int startIndex, Predicate<T> match);
```

Figure 2. Definition of List class.

The goal of this chapter to introduce the classes. To continue with the rest of the activities, please create a new project called Module9_yourName.

1) Creating classes in C#

Classes determine the characteristics of objects in terms of the data that objects can hold (fields and properties) and the actions that objects can perform (methods). Imagine that you want to create circle objects in your program. For this, it is better you create a class named `Circle`.

We will define the Circle class. Before we do that, we will create a new folder named Model. I recommend adding all classes to this folder since by creating the classes we are actually building the model of our program. To create a new folder, please right click on the project name in the solution explorer. A popup menu will appear. Choose **Add** -> **New Folder** as shown in Figure 3.

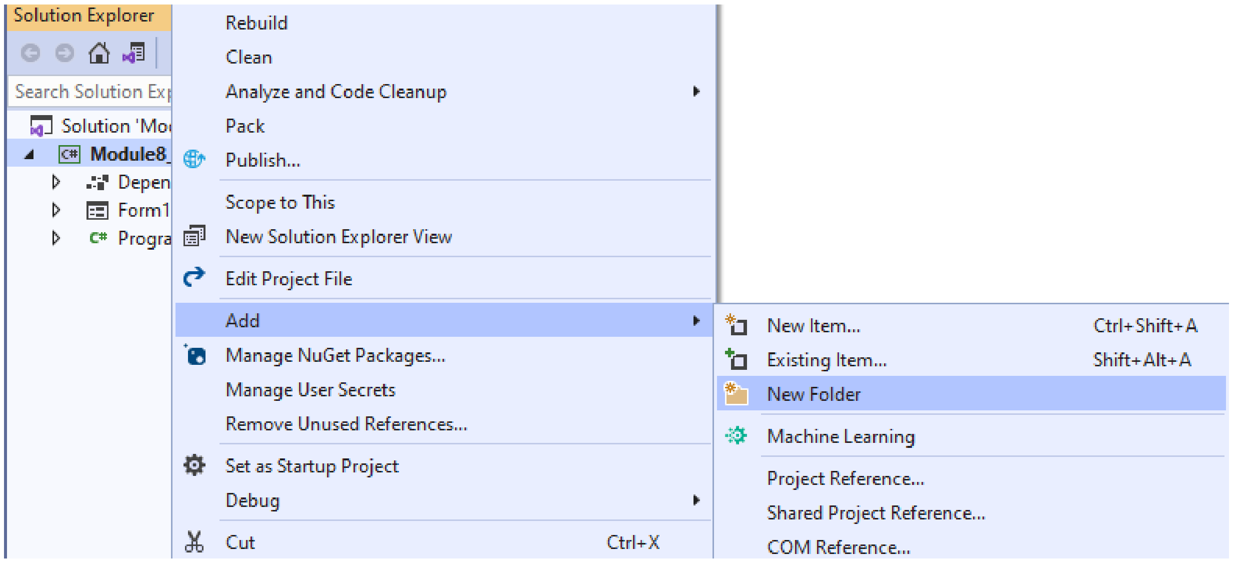


Figure 3. Adding a new folder to the project.

Last, please name the folder as Model. Your Solution Explorer window should look similar to Figure 4.

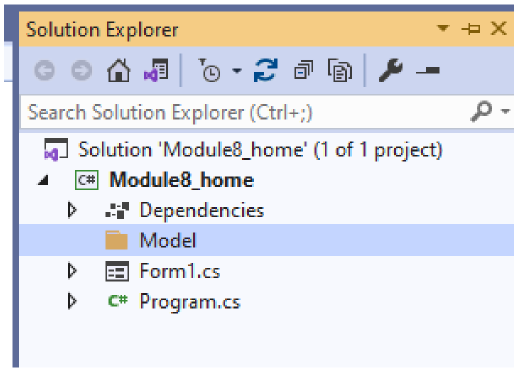

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Module8_home' (1 of 1 project)
Module8_home
Dependencies
Model
Form1.cs
Program.cs

Figure 4. Naming the new folder.

Now that we have the Model folder created, we will create a new class file named **Circle**. To do that, please right click on Model folder, choose *Add* and then *Class* from the popup menus. This process is visualized in Figure 5.

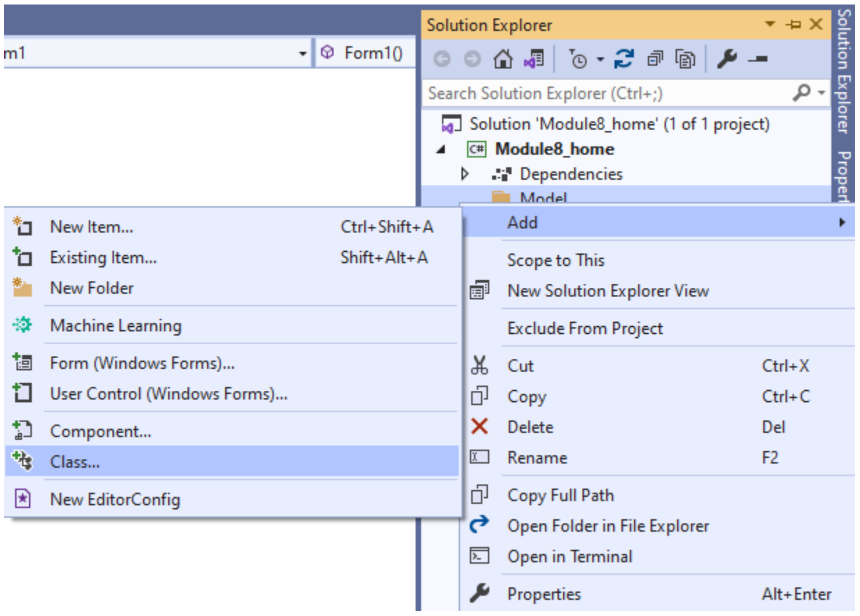


Figure 5. Adding a new class to Model folder.

A dialog window should appear where you need to enter a name for the class file. We will name our class as **Circle** as shown in Figure 6.

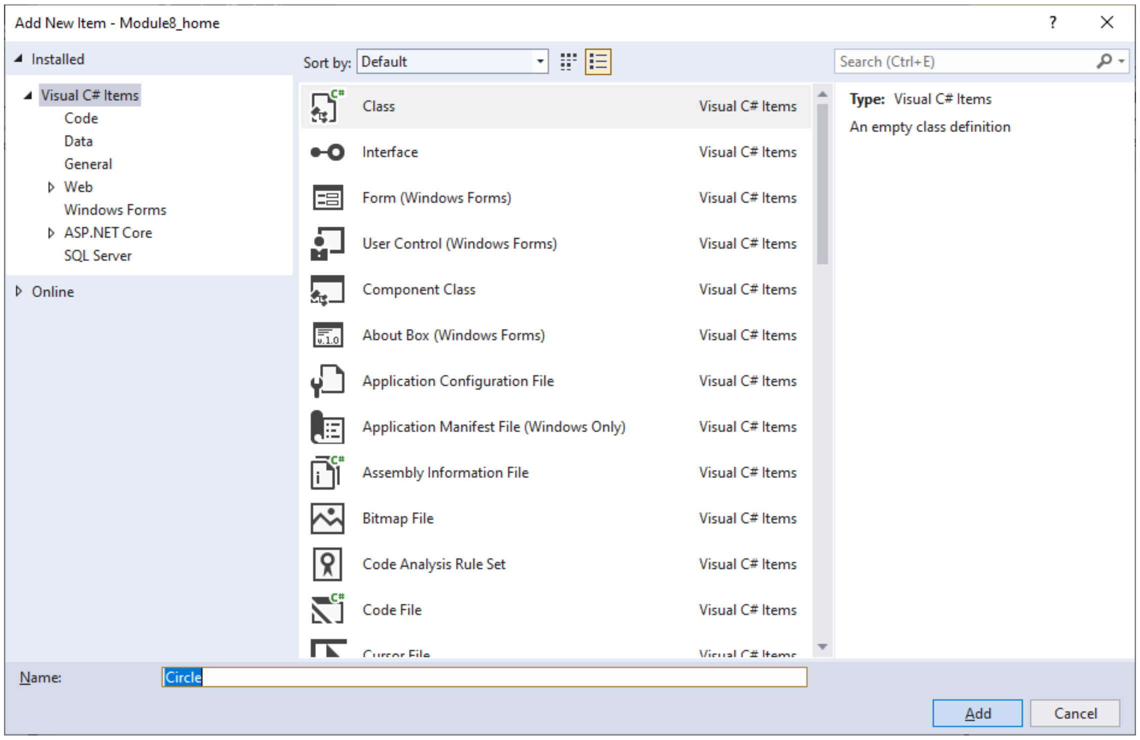
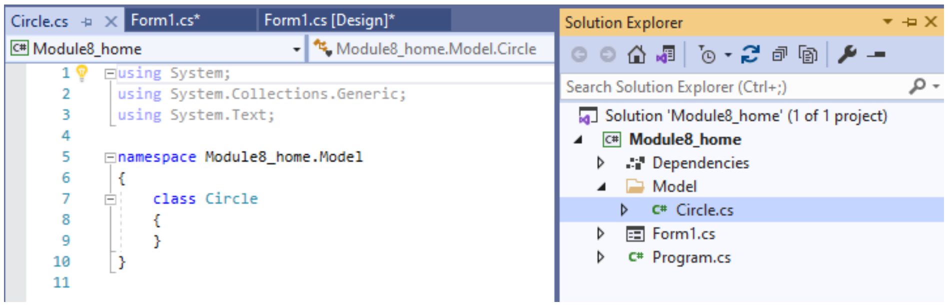

Add New Item - Module8_home
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
ASP.NET Core
SQL Server
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C# Items
Interface Visual C# Items
Form (Windows Forms) Visual C# Items
User Control (Windows Forms) Visual C# Items
Component Class Visual C# Items
About Box (Windows Forms) Visual C# Items
Application Configuration File Visual C# Items
Application Manifest File (Windows Only) Visual C# Items
Assembly Information File Visual C# Items
Bitmap File Visual C# Items
Code Analysis Rule Set Visual C# Items
Code File Visual C# Items
Type: Visual C# Items
An empty class definition
Name: Circle
Add
Cancel

Figure 6. Creating a new class named Circle.

After pressing the Add button, you should have the Circle.cs (which is the source code file of the class) added inside the Model folder. The source code file will be automatically opened as seen in Figure 7 below.



```
using System;
using System.Collections.Generic;
using System.Text;

namespace Module8_home.Model
{
    class Circle
    {
    }
}
```

Figure 7. Circle class is created.

Initially, your class will be empty, meaning that it will not have any properties or methods defined. As shown in Figure 8, class definition basically consists of a header and body. In header `class` keyword is accompanied with the name of the class (e.g., `class Circle`). You should always use uppercase for the first letter of the class name.

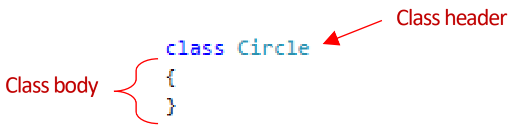


Figure 8. Class definition.

Once a class is defined, you can create new objects from that class. For example, in the following code, we create two Circle objects:

```
Circle smallCircle = new Circle();
Circle bigCircle = new Circle();
```

2) Defining class properties

The `Circle` class is now empty, and we will define it. A class definition consists of properties (and fields) and methods. A **property** defines a specific characteristic of a class and holds data pertaining to that characteristic. For example, a *Label* control has a *Text* property that is used to determine what text the label should display, or *ForeColor* property to change the font colour.

We will define a **diameter** property for the `Circle` class. A property works in conjunction with a **field**. Field holds the value for the object, while property allows us to set or get the value of the field. A property contains two special types of methods, called **accessors**, which are `get` and `set`. While `get` is invoked when we read the property value, `set` is called when we assign a new value to the property.

A field is always defined as private and should not be accessible outside the class. Since they hold critical data about objects, allowing direct access to them is not recommended. This is a common case, and you should follow it (although it is possible to define them as public). Encapsulating them inside the accessor methods, provides a more secure and effective control over the private fields.

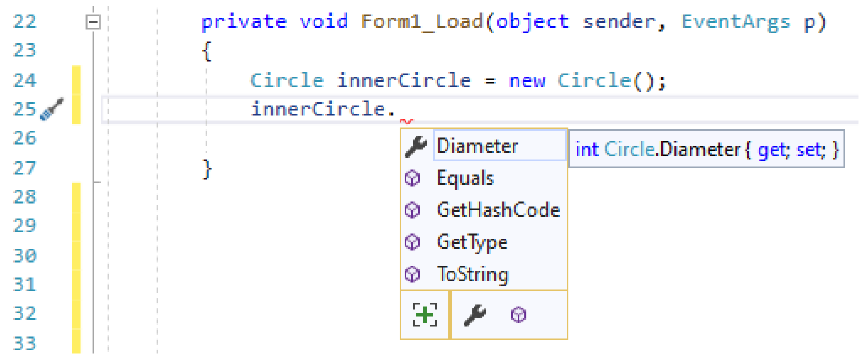
The code below shows how to define a property, called **Diameter**, for the `Circle` class. We first define a private field, which holds the value for this property. Typically, the field name is determined by appending _ in front of the property name. In our case, the field name will be _*diameter*. Then, the **Diameter** property is defined using set and get accessors. While `get` returns the _diameter field, `set` will update the _diameter field with some value.

```
class Circle
{
  private int _diameter;

  public int Diameter
  {
    get { return _diameter; }
    set { _diameter = value; }
  }
}
```

value is an "implicit" parameter because it is automatically created by the compiler. It carries the data type of the property. In this case, the **value** parameter's data type is **int** .

Now, we can create objects from the `Circle` class. The code at line 24 in Figure 9 creates a `Circle` object called `innerCircle`. "`Circle innerCircle`" is the expression on the left hand-side of "`=`". This declares a variable that will hold a reference to the object. `new Circle()` is the expression on the right hand-side of "**=**", which creates the `innerCircle` object in the memory and returns a reference to it by calling the **constructor** of the class using the *new* keyword. We will cover constructors soon in this module.

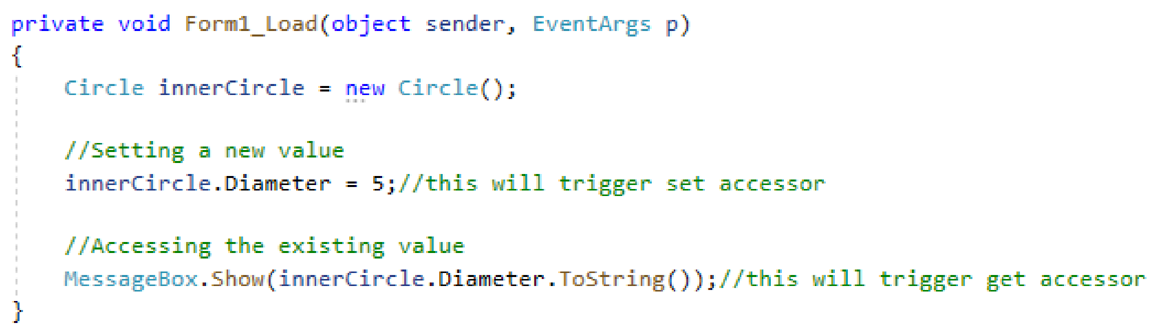
When you place a dot next to the object name (see line 26 in Figure 9), you should be able to see the property name called **Diameter** displayed in a list, but not the **_diameter** field. This is because the Diameter property was declared as `public`, whereas the _diameter field was declared as `private`. Private class members are NOT accessible through the objects.



```
22          private void Form1_Load(object sender, EventArgs p)
23          {
24              Circle innerCircle = new Circle();
25              innerCircle.
26
27          }
28
29
30
31
32
33
```

Figure 9. Class definition.

After creating the `innerCircle` object, we will set the value of the Diameter property to 5. To do that, we basically use the assignment operator =. This is similar to setting a new value to the Text property of a label or textbox, which we have repeated many times. Next, we will print the **Diameter** value of the `innerCircle` object using *MessageBox.Show* method. The code necessary to perform these operations is provided below in Figure 10.

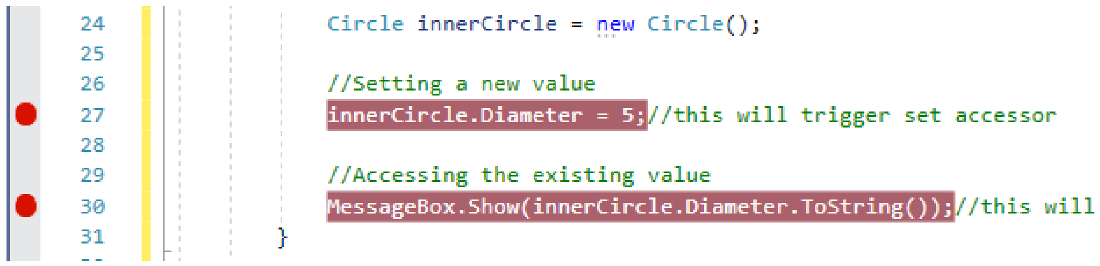


```
private void Form1_Load(object sender, EventArgs p)
{
    Circle innerCircle = new Circle();

    //Setting a new value
    innerCircle.Diameter = 5;//this will trigger set accessor

    //Accessing the existing value
    MessageBox.Show(innerCircle.Diameter.ToString());//this will trigger get accessor
}
```

Figure 10. Using the Diameter property of the innerCircle object.

You can create as many new object as you want using the `Circle` class. All these objects would have a **Diameter** property, which can have a different value for each distinct object.

We will test how our program functions by adding some breakpoints. First, please add the following breakpoints inside the main source code file.



```
        Circle innerCircle = new Circle();

        //Setting a new value
●       innerCircle.Diameter = 5;//this will trigger set accessor

        //Accessing the existing value
●       MessageBox.Show(innerCircle.Diameter.ToString());//this will
    }
```

Figure 11. Adding break points inside the form load event.

Next, we will add some breakpoints inside the Circle class. To navigate to the class file, please do a right-click on the Circle, and choose **Go To Definition** from the popup menu (or press F12).

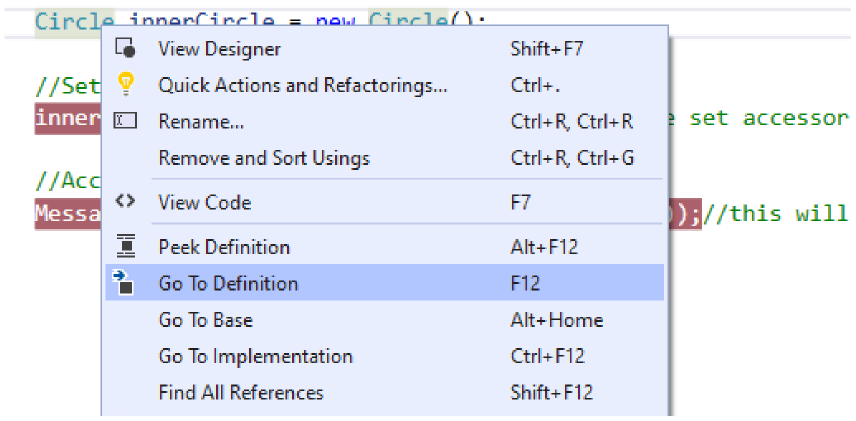
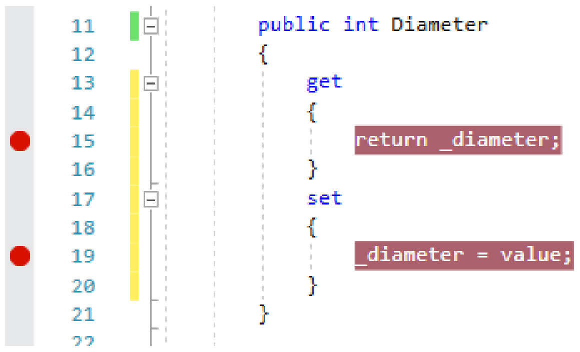

View Designer Shift+F7
Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
View Code F7
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Shift+F12

Figure 12. Popup menu to jump to the class definition.

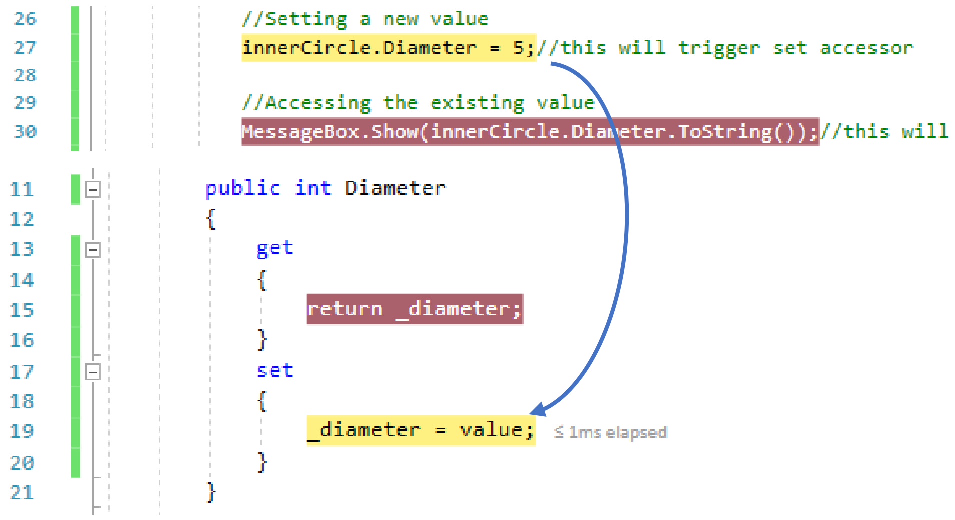
Then, add the following breakpoints inside the class file.



```
public int Diameter
{
    get
    {
        return _diameter;
    }
    set
    {
        _diameter = value;
    }
}
```

Figure 13. Adding break points inside the class definition.

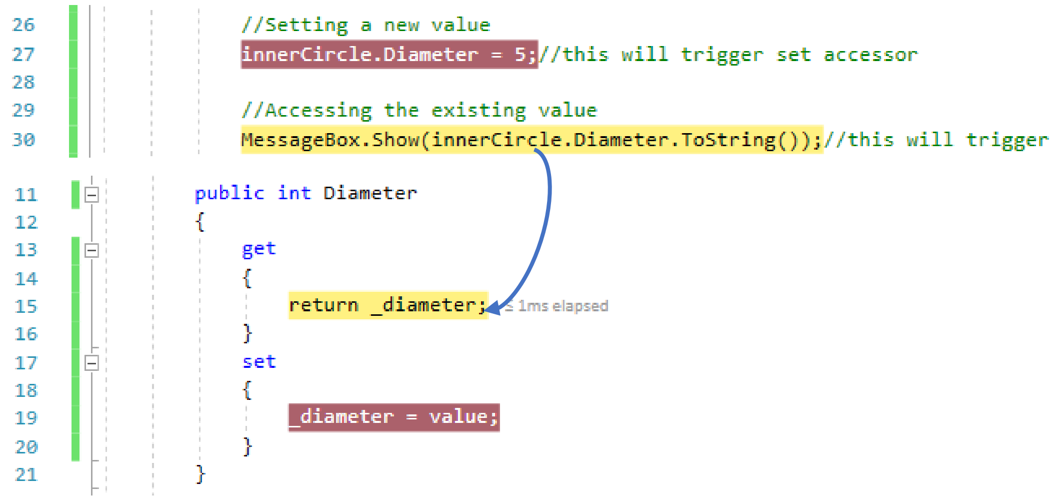
Now, please run your application by pressing F5.



```
//Setting a new value
innerCircle.Diameter = 5;//this will trigger set accessor

//Accessing the existing value
MessageBox.Show(innerCircle.Diameter.ToString());//this will
```

```
public int Diameter
{
    get
    {
        return _diameter;
    }
    set
    {
        _diameter = value;  ≤ 1ms elapsed
    }
}
```

Figure 14. set accessor is called when setting a new value.



```
//Setting a new value
innerCircle.Diameter = 5;//this will trigger set accessor

//Accessing the existing value
MessageBox.Show(innerCircle.Diameter.ToString());//this will trigger
```

```
public int Diameter
{
    get
    {
        return _diameter;  ≤ 1ms elapsed
    }
    set
    {
        _diameter = value;
    }
}
```

Figure 15. get accessor is called when the value is accessed.

As shown in Figure 14, the program should first pause at line 27 in the main source code file, and then at line 19 in the class source code file. This is because `set` accessor is executed when assigning a value to a property of an object. Next, as shown in Figure 15, the program should pause at line 30 in the main source code file, and then at line 15 in the class source code file. This is because `get` accessor is executed when reading a value of an object property.

3) Defining read-only properties

We will create a new property called **Radius** for the `Circle` class. The code below shows the definition of this property. The get accessor of this property returns `_diameter / 2`. However, it does NOT have the `set` accessor defined, which makes this property **read-only**. That is users cannot change the Radius value of the `Circle` objects.

```
class Circle
{
  private int _diameter;

  public int Diameter
  {
    get { return _diameter; }
    set { _diameter = value; }
  }

  public double Radius
  {
    get { return _diameter / 2; }
  }
}
```

One a property value is highly dependent on another property (e.g., radius is half of diameter), you should always make the dependent property (e.g., Radius) read-only. Allowing the user to change the value of a dependent property will result in incorrect values and errors.

4) Using auto property.

When a property simply sets and gets the value of a field, as the Diameter property does, the code can be simplified using auto property. By using auto-properties you can simplify the code for creating properties by NOT declaring a backing field, and by NOT writing code to get and set the property’s value. Below is the definition of the Diameter property as an auto-property.

```
public int Diameter
{
  get;
  set;
}
```

When auto property is used, a hidden backing field as well as the code for the get and set methods are automatically created by the compiler. Actually, I most of the time define the properties using the following short syntax:

```
        public int Diameter { get; set; }
```

4) Defining methods for classes.

You can also define methods for classes. For example, for `Circle` class, we can define two methods. One to calculate the diameter of the circle, and another one to calculate the area of the circle. The definition of these methods (`CalculatePerimeter` and `CalculateArea`) is provided below.

```
    public double CalculateArea()
    {
      return Math.PI * Radius * Radius; //Math.Pow(Radius,2)
    }

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

These methods basically apply the related mathematical formula to compute the area and diameter of a circle based on the value stored in the property **Radius**. π is obtained using the `Math.PI` constant. These methods for computing area and perimter will be available for any instances of the `Circle` class.

We will create a simple application to compute diameter and area of a circle object whose diameter is provided by the user. As shown in Figure 16, the interface is composed of:

- a textbox named `txt_diameter` where user needs to enter the diameter value,
- a label named `lbl_output` to print the computed diameter or area values,
- a button named `btn_computePerimeter` to compute the diameter, and
- a button named `btn_computeArea` to compute the area of the circle.

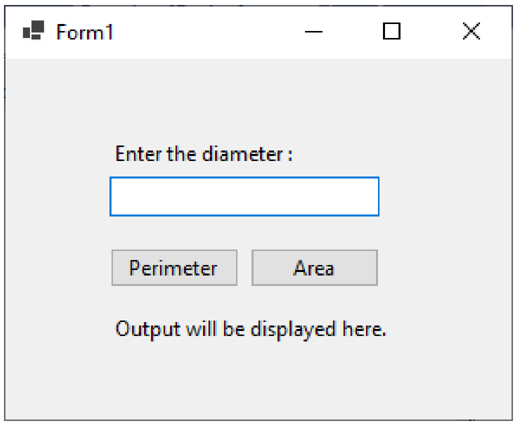


Figure 16. Interface of a simple application to compute diameter and area.

We will implement the click event handlers for both buttons. Please double click on the *Perimeter* button to create its click event handler. Inside there, we will create a new `Circle` instance called `myCircle`. We will convert the user input into integer and set it to the **Diameter** property of `myCircle`. Last, we will call the **CalculatePerimeter** property to calculate the perimeter of `myCircle` and print it in `lbl_output`. The complete code is shown below in Figure 17.

```
private void btn_computePerimeter_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();
    myCircle.Diameter = int.Parse(txt_diameter.Text);
    lbl_output.Text = "Perimeter : " + myCircle.CalculatePerimeter().ToString();
}
```

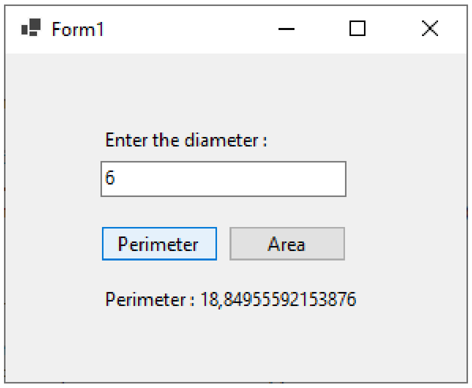
Figure 17. Click event handler for btn_computePerimeter.

Similarly, we will implement the click event handler for `btn_computeArea`. The code will be the same except that we will need to call the **CalculateArea** method this time. The code is shown in Figure 18.

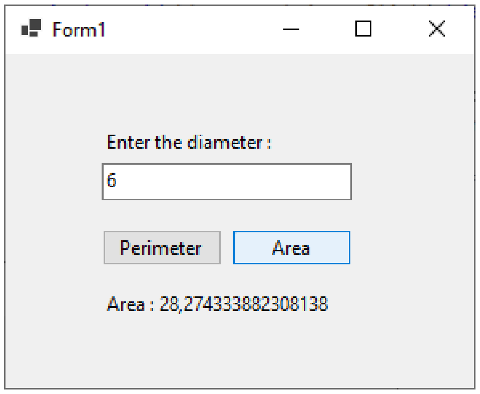
```
private void btn_computeArea_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();
    myCircle.Diameter = int.Parse(txt_diameter.Text);
    lbl_output.Text = "Area : " + myCircle.CalculateArea().ToString();
}
```

Figure 18. Click event handler for btn_computeArea.

Now, you can test your application. Figure 19 provides some sample screens from the running application.



```
Perimeter : 18,84955592153876
```



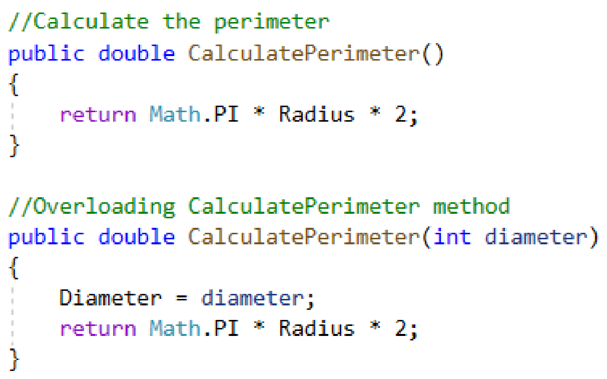
```
Area : 28,274333882308138
```

Figure 19. Sample screens from the running application.

5) Overloading methods.

In C#, you can define the same method but with <u>different</u> parameters. This process is called **overloading** and it results in **overloaded** methods. For example, neither CalculatePerimeter nor CalculateArea methods accepts a parameter. We can *overload* these methods so that they can optionally accept a new diameter value. In this way, instead of setting the diameter to some initial values, we can choose to directly pass the diameter value to these methods for calculation.

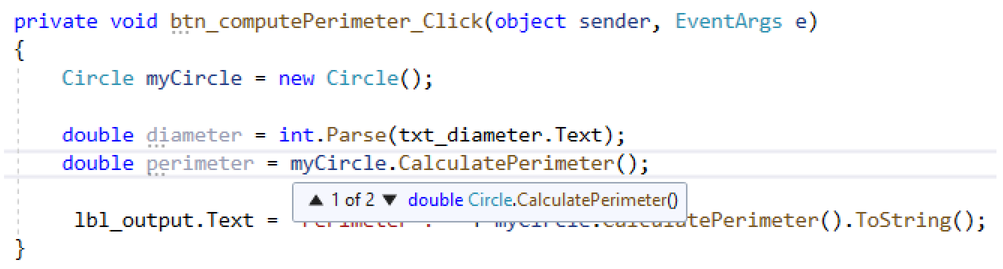
The following code overloads the CalculatePerimeter method. In other words, it creates a second version of the same method that accepts diameter parameter.



```
//Calculate the perimeter
public double CalculatePerimeter()
{
    return Math.PI * Radius * 2;
}

//Overloading CalculatePerimeter method
public double CalculatePerimeter(int diameter)
{
    Diameter = diameter;
    return Math.PI * Radius * 2;
}
```

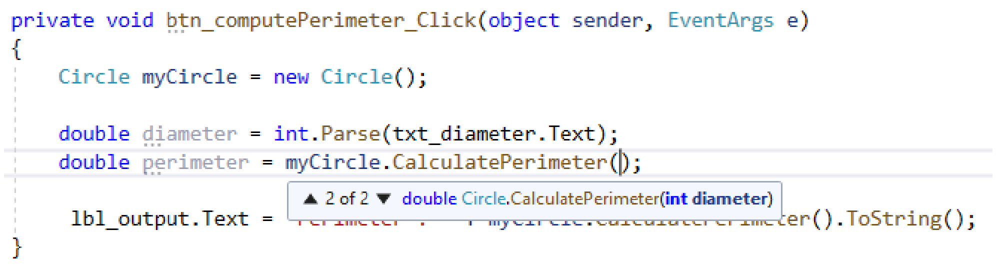
Figure 20. Overloading the CalculatePerimeter method.

Now, we can update our existing code to use the overloaded method to calculate the perimeter. When you write the code to call the CalculatePerimeter of myCircle object, the possible parameter options will be displayed in a tooltip as shown in Figure 21 and Figure 22. If you click on the small up or down arrows you can switch between different options that you can use. Since we have overloaded our method once, you should be able to see two options.



```
private void btn_computePerimeter_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();

    double diameter = int.Parse(txt_diameter.Text);
    double perimeter = myCircle.CalculatePerimeter();
              ▲ 1 of 2 ▼ double Circle.CalculatePerimeter()
     lbl_output.Text = "Perimeter: " + myCircle.CalculatePerimeter().ToString();
}
```

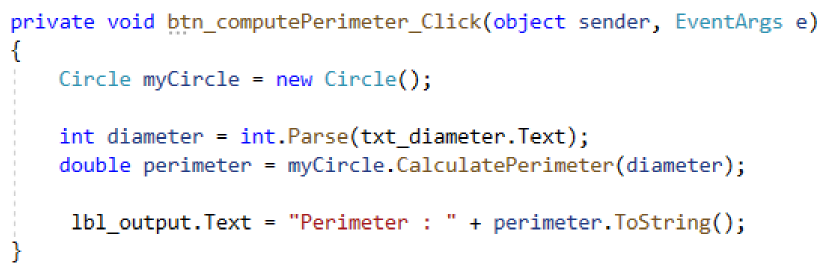
Figure 21. Viewing the parameter options for the CalculatePerimeter method.



```
private void btn_computePerimeter_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();

    double diameter = int.Parse(txt_diameter.Text);
    double perimeter = myCircle.CalculatePerimeter();
              ▲ 2 of 2 ▼ double Circle.CalculatePerimeter(int diameter)
     lbl_output.Text = "Perimeter: " + myCircle.CalculatePerimeter().ToString();
}
```

Figure 22. Viewing the parameter options for the CalculatePerimeter method.

We will use the overloaded method, which means we will pass the diameter to the CalculatePerimeter. The sample code is shown in Figure 22.



```
private void btn_computePerimeter_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();

    int diameter = int.Parse(txt_diameter.Text);
    double perimeter = myCircle.CalculatePerimeter(diameter);

     lbl_output.Text = "Perimeter : " + perimeter.ToString();
}
```

Figure 23. Using the overloaded CalculatePerimeter method.

In runtime, the compiler chooses which version of the method to run by matching the method call with the method **signature**. Method signature consists of the method name and the type of the parameters passed. You CANNOT overload a method by ONLY changing its return type.

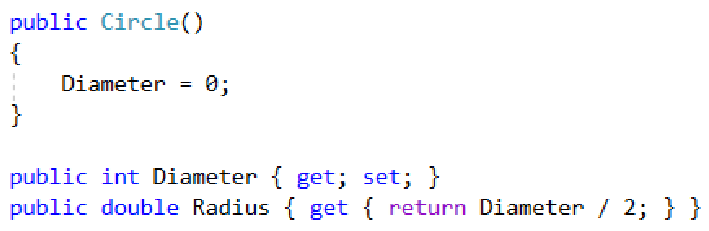
For example, the following code would throw an error since two methods have the same signature although they return a different data type.

```
//Calculate the area
public double CalculateArea()
{
  return Math.PI * Radius * Radius;
}

//Overloading CalculateArea
public int CalculateArea()
{
  return int.Parse(Math.PI * Radius * Radius);
}
```

6) Constructors

We can define constructors for classes to set some initial values for the properties. For example, the Diameter property does not have any initial value when a new `Circle` object is created. We can define a constructor that sets a Diameter to 0 when an object is created.

Constructors are actually methods that have the same name with the class. Below, we define a parameterless constructor for the `Circle` class. This constructor is parameterless since it does not accept any parameter by definition.



```
public Circle()
{
    Diameter = 0;
}

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

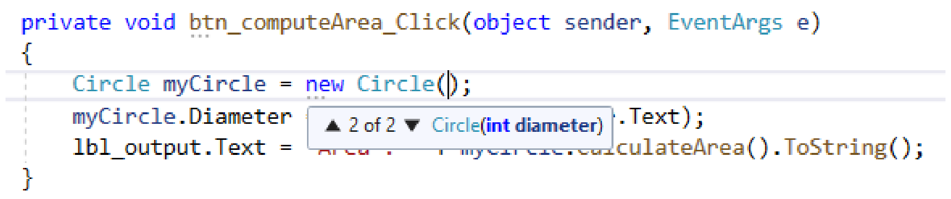
Figure 24. Defining a parameterless constructor method.

As we overload methods, we can also overload the constructors. Below in Figure 25, we overload the constructor to optionally accept an initial diameter value.

```
class Circle
{
    public Circle()
    {
        Diameter = 0;
    }

    public Circle(int diameter)
    {
        Diameter = diameter;
    }

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

Figure 25. Defining a parameterless constructor method.

Let's use the overloaded constructor when computing the area. When you define a new `Circle` instance, you should be able to see different constructors that you can use within a tooltip (see Figure 26).



```
private void btn_computeArea_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();
    myCircle.Diameter  ▲ 2 of 2 ▼ Circle(int diameter) .Text);
    lbl_output.Text =                  culateArea().ToString();
}
```

Figure 26. Using the overloaded constructor.

Below is the updated definition of `btn_computeArea_Click` handler to compute the area by using the *parameterized* constructor. The main difference is that we pass the diameter value when creating the `myCircle` object instead of assigning the value to the Parameter property after creating the object.

```
private void btn_computeArea_Click(object sender, EventArgs e)
{
    int diameter = int.Parse(txt_diameter.Text);
    Circle myCircle = new Circle(diameter);

    double area = myCircle.CalculateArea();
    lbl_output.Text = "Area : " + area.ToString();
}
```

Figure 27. Using the overloaded constructor.

It is totally okay NOT to create a constructor as we did initially in the chapter. In that case, a default parameterless constructor is automatically created by the compiler in runtime.

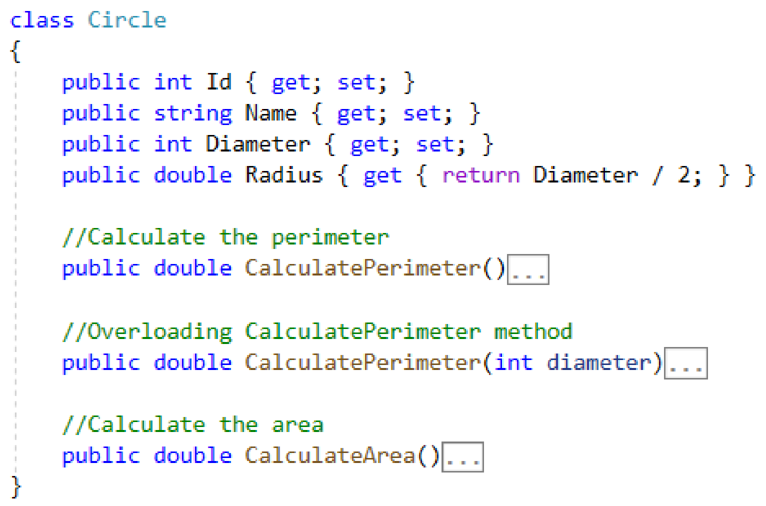
7) Using lists to store class type objects

As covered in previous modules, lists can be used to store a set of objects. We can create a `List` to hold objects of a specific class type, such as `Circle`. The following code defines a `List` called `circles` that can store a collection of circle instances.

```
        List<Circle> circles = new List<Circle>();
```

Notice that the word `Circle` is written inside angled brackets, <>, immediately after the word `List`. This indicates that the `circles List` can hold only objects of the `Circle` class type.

To add a new item to `circles`, we need to first define a `Circle` object. Then, we can use the **Add** method to add a new object to `circles`. Before adding a new item, we will update the `Circle` class definition as shown in Figure 28. In the updated definition, the constructors are removed, and Name and Id properties are added.



```
class Circle
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Diameter { get; set; }
    public double Radius { get { return Diameter / 2; } }

    //Calculate the perimeter
    public double CalculatePerimeter()...

    //Overloading CalculatePerimeter method
    public double CalculatePerimeter(int diameter)...

    //Calculate the area
    public double CalculateArea()...
}
```

Figure 28. The updated class definition.

The following code shows an example for adding a new `Circle` class type object to a list.

```
        //Create the circles list object
        List<Circle> circles = new List<Circle>();

        //Create a new instance of the Circle class type
        Circle circle1 = new Circle
        {
           Id = 1,
           Name = "Inner circle",
           Diameter = 10,
        };

        //Add the circle1 object to the list
        circles.Add(circle1);
```

7) Using listbox control to display class type objects

Listbox controls can be very convenient to display a set of class type objects. We will develop a new application (or change the existing one), in which the users will be able to create new `Circle` objects, them to a `List`, and display this list with a Listbox. Perimeter and area of the circle that is selected in the listbox will be automatically calculated. The application interface is displayed in Figure 29 below. Please name all controls as suggested in the figure.

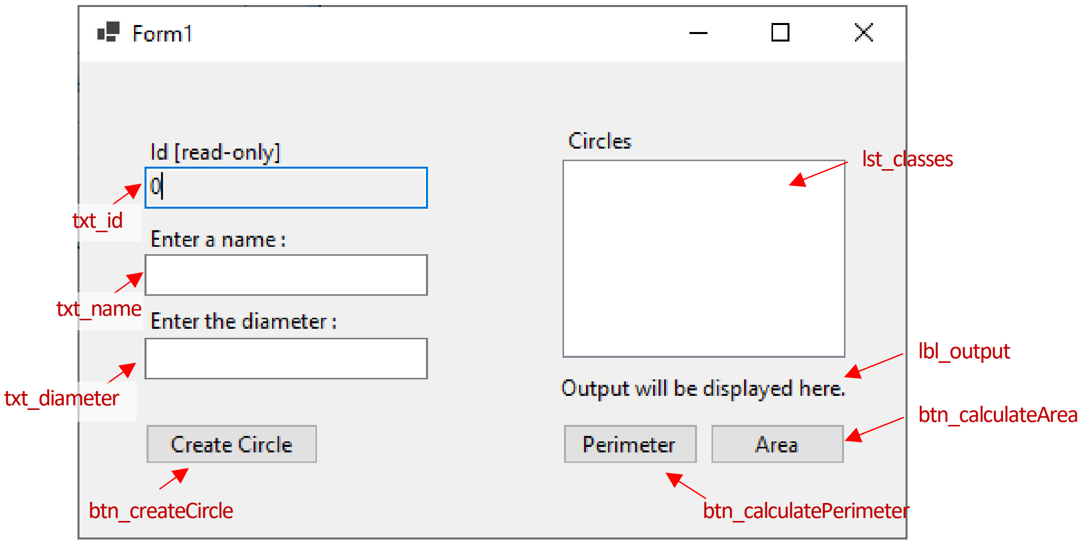


Figure 29. The application interface.

First, we will define a **`BindingList`** object called **circles** that will store the `Circle` type objects. **circles** will serve as the (*fake*) database for this application. `BindingList` can be considered as a special version of `List` and most commonly used when binding data to form controls that can show multiple records, such as ListBox.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    BindingList<Circle> circles = new BindingList<Circle>();
```

Figure 30. Defining the circles BindingList.

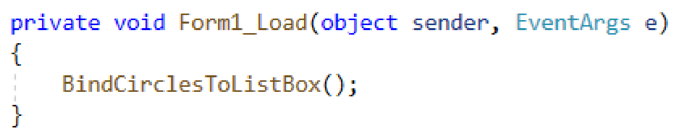
Next, we will define a method called **BindCirclesToListBox**. This method will bind `circles BindingList` object to lst_circles. This is done by assigning `circles` to **DataSource** property of `lst_circles`.

```
private void BindCirclesToListBox()
{
    lst_circles.DataSource = circles;
    lst_circles.ValueMember = "Id";
    lst_circles.DisplayMember = "Name";
}
```

Figure 31. Defining BindCirclesToListBox method.

We bind the data but right now `lst_circles` does not know which properties of `Circle` type objects to use for displaying and identifying the items. In the listbox, we want to display the **Name** property of the circle objects. For this purpose, we set the DisplayMember property of lst_circles to “Name”. Last, we want to identify the selected item in the listbox by using the Id property of the circle objects. To do this, we set ValueMember to “Id”. These two properties accept a string value which should be equal to a property of the class type objects being listed.

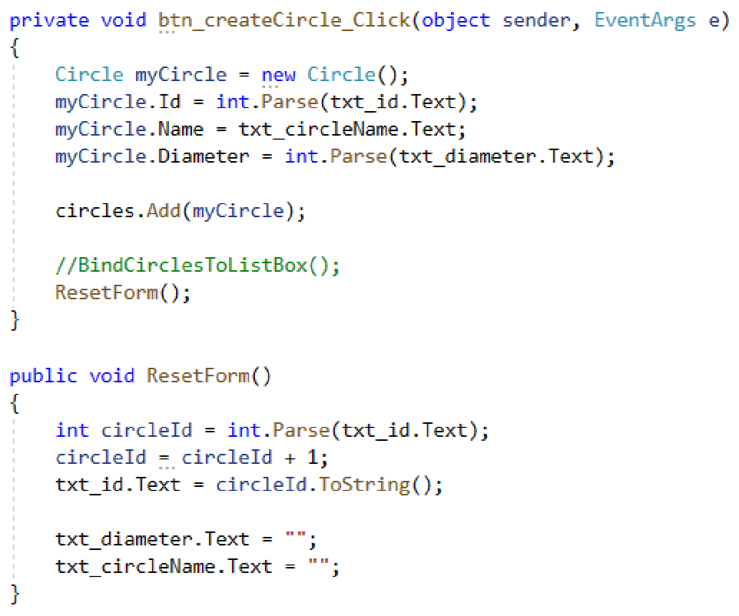
As shown below, **BindCirclesToListBox** will be called only once when the form is loaded. Any changes made to `circles`, will be automatically reflected to `lst_circles`.



```
private void Form1_Load(object sender, EventArgs e)
{
    BindCirclesToListBox();
}
```

Figure 32. Defining BindCirclesToListBox method.

Now, we will implement the click event handler for the `btn_createCircle` button (see Figure 33). By using the circle name and diameter values provided by the user, we will create a new instance of `Circle`, called `myCircle`. The Id of `myCircle` will be set to the value of `txt_id.Text`, which is initially 0. Then, we will add `myCircle` to `circles` list by using the **Add** method. Any changes to `circles` will be automatically reflected in the listbox since `circles`, which is a BindCirclesToListBox type, was once bounded to the listbox `lst_circles`. Therefore, there is NO need to call BindCirclesToListBox method again.



```
private void btn_createCircle_Click(object sender, EventArgs e)
{
    Circle myCircle = new Circle();
    myCircle.Id = int.Parse(txt_id.Text);
    myCircle.Name = txt_circleName.Text;
    myCircle.Diameter = int.Parse(txt_diameter.Text);

    circles.Add(myCircle);

    //BindCirclesToListBox();
    ResetForm();
}

public void ResetForm()
{
    int circleId = int.Parse(txt_id.Text);
    circleId = circleId + 1;
    txt_id.Text = circleId.ToString();

    txt_diameter.Text = "";
    txt_circleName.Text = "";
}
```

Figure 33. Adding a new circle.

After the new circle is added, we need to reset the form to let the user enter a new circle. For this purpose, we create a new method called **ResetForm** and call this method inside the click event handler. ResetForm, will clear the content of `txt_diameter` and `txt_circleName`, and more importantly it will increase the value of `txt_id.Text` value by one. This updated `txt_id.Text` value will be the Id of the next circle instance to be created. The implementation of ResetForm is shown in Figure 33.

We will test that we have built so far. Please run your application by pressing F5. Enter the values provided in Figure 34 and click on the Create Circle button.

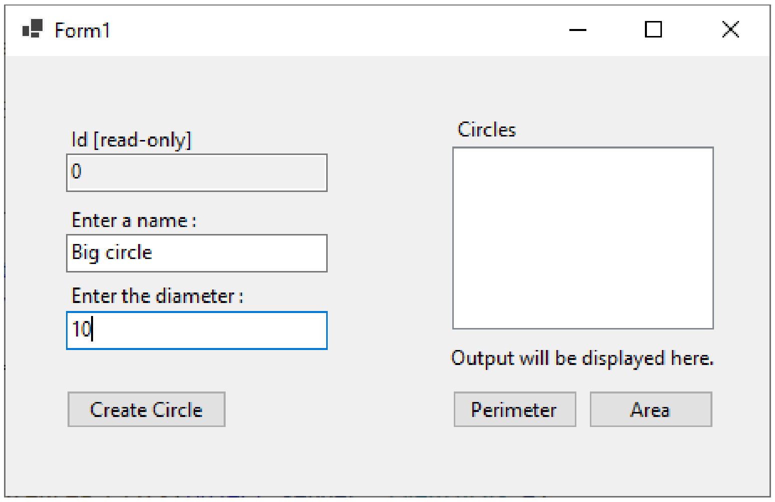


Figure 34. Creating a new circle.

In the next screen (see Figure 35), you should see the “Big circle” added to the listbox. Additionally, the form should be reset, and the Id should be increased to 1.

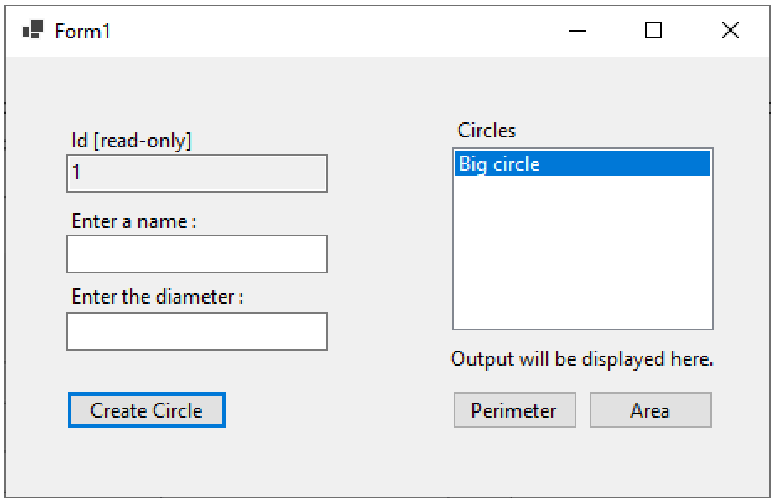


Figure 35. New circle is added, and the form is reset..

Now, we will implement the click event handler for `btn_computePerimeter` and `btn_computeArea`. These buttons should calculate the perimeter or area for a selected circle instance in the listbox.

We will first implement `btn_computePerimeter_Click`. To calculate the perimeter, we need to know which circle instance is selected. As you may remember, circles are identified by the **Id** property, and this property was set as the **ValueMember** of the listbox when the `circles` list was bounded to the listbox.

This means we should be able to access the Id of any selected item through the **SelectedValue** property. In the code shown in Figure 36, `lst_circles.SelectedValue` was used to read the Id of the selected circle item in the list and assigned to `circleId`.

Once we know the Id of the selected circle, we use the **Single** method to return the only single circle object from `circles` whose Id is equal to `circleId`. The returned circle object is assigned to `selectedCircle`. Then, all we need to do is to call **CalculatePerimeter** method `selectedCircle` and print the result in `lbl_output`. See Figure 36 for the complete implementation.

```
private void btn_computePerimeter_Click(object sender, EventArgs e)
{
    int circleId = (int)lst_circles.SelectedValue;
    Circle selectedCircle = circles.Single(c => c.Id == circleId);

    double perimeter = selectedCircle.CalculatePerimeter();
    lbl_output.Text = "Perimeter : " + perimeter.ToString();
}
```

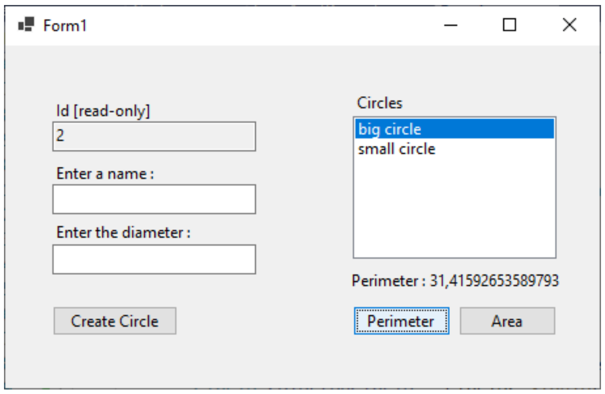
Figure 36. Calculating the perimeter of the selected circle.

Last, we will implement the click event handler for btn_computeArea. We can use almost the same code here. Only difference is that we need to call **CalculateArea** method for the selected circle. See Figure 37.

```
private void btn_computeArea_Click(object sender, EventArgs e)
{
    int circleId = (int)lst_circles.SelectedValue;
    Circle selectedCircle = circles.Single(c=>c.Id== circleId);

    double area = selectedCircle.CalculateArea();
    lbl_output.Text = "Area : " + area.ToString();
}
```

Figure 37. Calculating the area of the selected circle.

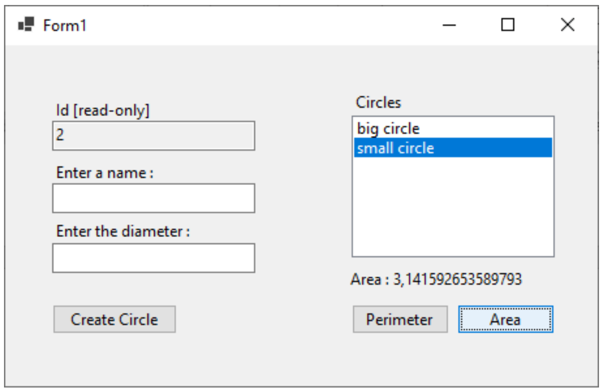
Now you can test your finished project. Add some circles, and then click on Perimeter or Area buttons. Some sample screens are shown below.



Id [read-only]: 2

Circles: big circle, small circle

Perimeter : 31,41592653589793



Id [read-only]: 2

Circles: big circle, small circle

Area : 3,141592653589793