Programming Using C#, Basic Course

Assignment 6b
OOP - Inheritance

Universal Calculator

Mandatory (for VG or A, B)

Farid Naisan
University Lecturer
Department of Computer Sciences
Malmö University, Sweden

Assignment 6- Inheritance

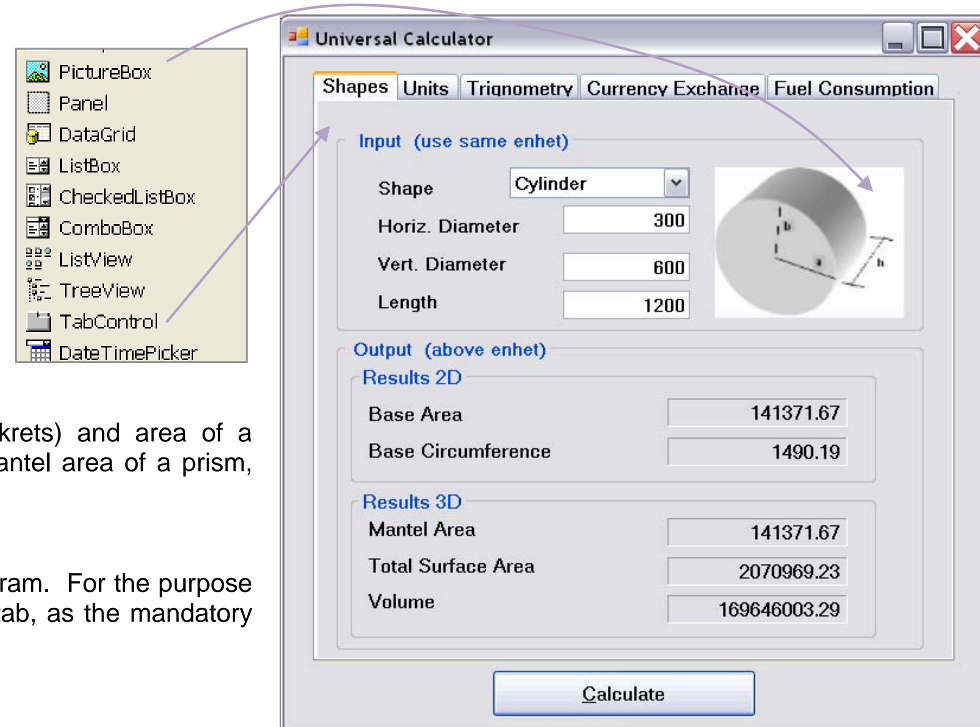
1. Objectives

In the last module, we practiced with the first of the three important aspects of OOP, encapsulation. Now it is time to move forward and try the next concept, inheritance. Object inheritance helps to reuse code by creating a **"is a"** relation between objects of same type. The following topics are covered in this assignment:

- inheritance techniques in C#,
- a couple of new Form controls, the **TabControl** and the **ImageList** control, and

2. Description

Computation of geometrical shapes, areas and volumes, conversion between different currencies and so on are tasks that we need in our everyday life. To make life easier, we will write a program where we can gather all such simple but useful calculations. In this assignment we will try to design and implement a part of this idea. We begin with the calculation of the geometrical shapes, volumes, surface areas (mantel areas, Swedish (SV: sidoytan) and periphery (SV: omkrets) of some standard shapes.



Write a program that calculates the circumference (SV: omkrets) and area of a rectangle, circle and ellipse as well as the volume and the mantel area of a prism, cylinder and a sphere.

The figure below presents a suggested GUI and a sample program. For the purpose of this assignment, it is sufficient to work only on the Shapes-tab, as the mandatory part, (se the figure below) and ignore the other ones.

3. Requirements and grading:

This assignment will be graded as VG, G or A,B, C (ECT) or will be subject to complementary work and resubmission.

3.1 The program must work satisfactorily as in the last assignment.

3.2 All class definitions, and class methods are to be documented using a XML-compatible format.

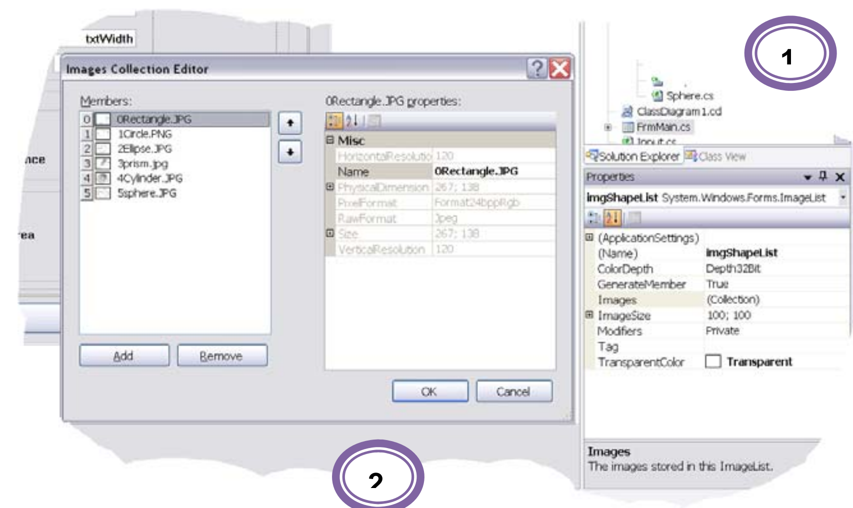
3.3 Auto-implemented properties (e.g. `public string Name { get; set; }`) are not allowed in this assignment. Put some control of the "`value`" in the `set`-properties to control its validities, wherever applicable.

3.4

3.5 You are expected to seriously apply encapsulation and inheritance. Do not use your own setter and getter methods; use properties instead whenever access to an instance variable is necessary. The instance variables that need not to be accessed require no property.

3.6 Use a **TabControl** with a couple of tabs (for training). Your work will be concentrated only in one of the tabs as only the shapes tab is the obligatory part of this assignment.

3.7 The shape-pictures in this document are downloadable as image files from the Module. Save the images in a sub-folder in your project.



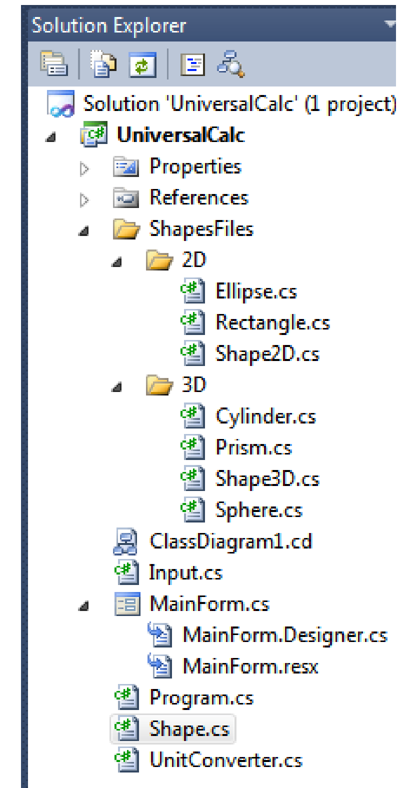
3.8 Use an **ImageList** control on the form and name it for example **imgShapeList**. The conrol will residet at the bottom part of the form as this is an "invisible" control like the Timer control. Go to the Property Window for this control and click on the Collection item. Use the Add button, browse to the sub-folder where you have saved the images and open a file. Do this for all images. Note that the image files are saved in at least two formats, jpg and png. ImageList supports several image formats.

3.9 Note that after putting the image files into your project, you don't need the files anymore, and in case you make any changes in the files, you have to remove the corresponding item from your ImageList and add the file again. The images will be embedded into resx file.

3.10 Use a PictureBox control to which a picture from the ImageList will be attached based on the user's choice of the shape. The PictureBox has a property named Image. If you have saved the images in the same order as the contents of the shape-ComboBox, you can use the following simple statement to show the correct picture:

```
picShape.Image = this.imgShapeList.Images[cmbShape.SelectedIndex];
```

4. The Project

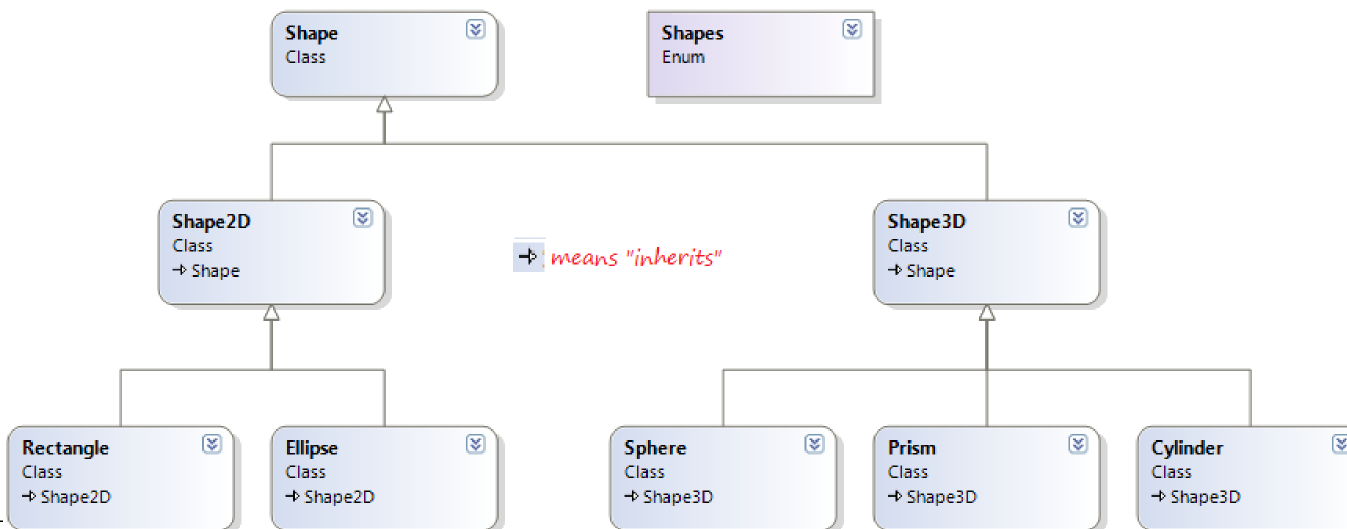


For your convenience, a sample project and a class diagram are presented in the following figures. The project figure show how you can organize your classes in a logical and physical groups. To create the folder ShapeFiles, right-click on the project (UniversalCalc) and then select Add - New Folder. To create the sub-folders 2D and 3D folder, right-click on the ShapeFiles and add new folders. When you add a new class in a folder, right-click on the folder name and then select Add – Class.

When you add a new folder in the Project Explorer (logical structure), VS creates the folder with the same hierarchy on your harddrive (physical structure). Organizing files and components in a project is a part of good programming style.

5. Class Diagram

The class diagram below illustrates how the classes are associated with each other, as far as inheritance is concerned.



Actually each of the 3D classes, Sphere, Prism and Cylinder can declare an instance of the Ellipse, Circle and Rectangle classes, respectively, for its base shape. For example, en cylinder has an ellipse as its base. In this case an object of the cylinder class has an aggregation relationship with an object of the Ellipse class. It is said to have a “has a” relationship. The model below displays even the aggregation relationships.

Shape
Class

has

Shapes
Enum

Shape2D
Class
→ Shape

Shape3D
Class
→ Shape

Rectangle
Class
→ Shape2D

Ellipse
Class
→ Shape2D

has

Sphere
Class
→ Shape3D

Prism
Class
→ Shape3D

Cylinder
Class
→ Shape3D

has

has

Think:

A Cylinder **is a** Shape3D

A Cylinder **has an** Ellipse

```
/// <summary>
///  Cylinder a shape that is like a prism but has elliptical
///  base form.
/// </summary>
public class Cylinder : Shape3D
{
    Ellipse baseShape;
```

Inheritance: "is a" relation

Aggregation: "has a" relation

6. The Shape class (Shape.cs)

6.1. This class defines shapes in general. It can contain fileds as name, pen color, pen thickness,etc, and methods that are common for all shapes in the program. It will serve as a base class for all the geometrical shapes. In this assignment, only one field and a minimum number of methods are used.

6.2. **Note**: It is more suitable to write this class as an **interface** type**,** but the subject is not within the scope of this course.

6.3. **Fields**: Declare a private field, **shapeType**, for defining the type of the figure which can be suitabley defined as a **public enum** and can be saved in the same file as Shape.cs.

```
/// <summary>
/// Autor:  Farid Naisan
/// The Enum Shapes lists those shapes that are included in the program.
/// </summary>
public enum Shapes
{
    Rectangle,
    Circle,
    Ellipse,
    Prism,
    Cylinder,
    Sphere
}
```

6.4. **Constructors**: Write a default constructor and let the **shapeType** variable have a default value **Shapes.Rectangle**.

6.5. **Properties:** get, set for **shapeType**.

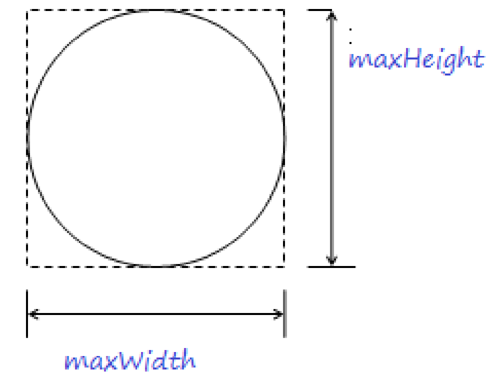
6.6. **Methods:**

6.6.1. Write a method which returns `true` if the figure is a three-dimensional shape (Prism, Cylinder, Sphere) and `false` otherwise.

6.6.2. Override the ToString method to return a string formatted with the name of the shape.

Note: Shape is a class and Shapes is an enum

7. The Shape2D class (Shape2D.cs)



The Shape2D class inherits the **Shape** class and is to serve as a base class to all two-dimensional figures. Normally, it is better to define such a class as an abstract class but this subject is not a part of this course.

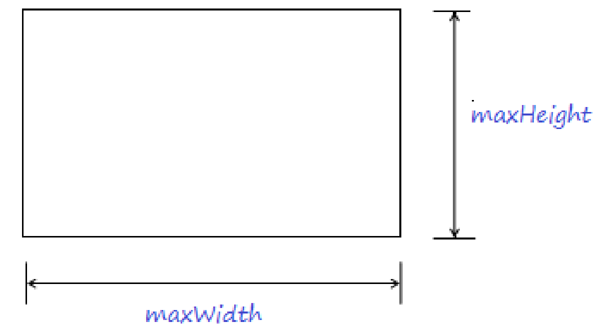
Fields: **maxWidth** (width) and **maxHeight** (heigt) of the bounding rectangle.

7.1. **Constructors**: write a constructor that takes maximum width and maximum height of the shape as parameters.

7.2. Properties: get, set functions for both of **maxWidth** and **maxHeight**.

7.3. **Methods:** Write a method **GetArea** which returns the product of **maxWidth** and **maxHeight**. Declare this function as virtual in order to make it possible for the sub-classes to override it, if necessary. You may have guessed from this discussion that virtual and override is a pair of modifiers that makes overriding possible. – use virtual in base class and ovveride in the sub-class with the metodh that you want to make overridable. What is overriding, by the way? The mehod ToString is virtual in its base ckass (object) in the .NET Framework.

8. The Rectangle class (Rectangle.cs)



8.1. The **Rectangle** class inherits the **Shape2D**.

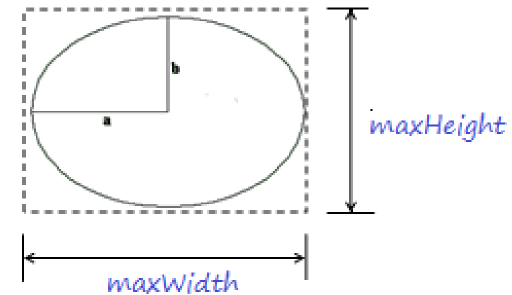
8.2. **Fields**: none.

8.3. **Constructors**: write a constructor that takes width and height of the shape as parameters. Set the **shapeType** of the base class to **Shapes.Rectangle** in the constructor. Note that the base class's constructor must be called.

8.4. **Properties**: get properties for the area and circumference of the shape returning a double value.

© Farid Naisan

9. The Ellipse class (Ellipse.cs)



9.1. The Ellipse class inherits the **Shape2D**. This class defines both circles and ellipses.

9.2. **Fields**: none.

9.3. **Constructors**:

9.3.1. Write a constructor that takes the horizontal and vertical radius of the ellipse as inparameters. For a circle, these two values are equal. Meanwhile, two times the redius gives the side of the enclosing rectangle. Set the shapeType of the base class **to Shapes.Ellipse** in the constructor.

9.3.2. Write another constructor with only radius as inparameter. This constructor can be called when creating a circle. Mek sure that the base class gets correct values for the **maxWidth** and **maxHeight**. Set the **shapeType** of the base class to **Shapes.Circle** in the constructor.

9.3.3. **Properties**: get properties for the area and circumference of the shape returning double values.

9.3.4. It must be pointed out that there isn't any simple way to calculate the circumference and area of an ellipse, but the following formulas can be used in this program:

```
circumference = 2*PI*sqrt((a2+b2)/2)

area = PI * a * b

maxWidth = 2 * a

maxHeight= 2 * b
```

where **a** and **b** are the radius in the x- and y-directions respectively. The formulas are valid for a circle as well setting **a = b**. Please make sure to avoid using short variable names as in the the formulas when programming. You can choose names like **xRadius**, **yRadius** instead of '**a**' and '**b**'.

10. The Shape3D class (Shape3D.cs)

10.1. The **Shape3D** class inherits the class **Shape** and is the base class to all the three-dimensional figures. Normally, it is better to define even this class as an abstract class but the subject is outside the scope of the course.

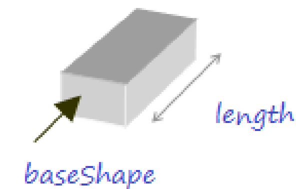
10.2. **Fields**: a double variable, **length**, for storing the value of the z-dimension of the 3D shape.

10.3. **Constructors**: a constructor with length as an in-parameter.

10.4. **Properties**: get and set properties for the length field.

10.5. **Methods**: none.

11. The Prism class (Prism.cs)



11.1. A prism is also called a rectangular solid or a cuboid. This class being a 3D figure inherits the class **Shape3D**.

11.2. **Fields**: **baseShape** of the **Rectangle** class type.

11.3. **Constructors**: Write a constructor that takes the width, height and length of the prism as in-parameters. Set the **baseShapes** values and pass the value of the shape's length to the base class's length field.

11.4. **Properties**: Write the following get-properties:

11.4.1. A property that returns the volume of the prism.

volume = baseArea * length

baseArea = a * b (i.e the area of baseShape)

11.4.2. A property that returns the mantel area (area of the sides) of the prism.

mantelArea = cirmferenceOfBaseShape * length

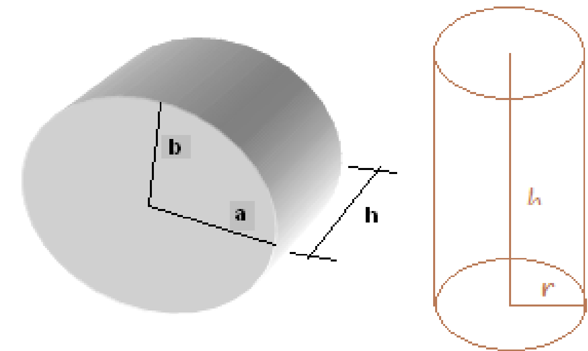
11.4.3. A property that returns the total surface area of the prism.

totalArea = mantelArea + 2*baseArea

11.4.4. A property that returns a copy of the **baseShape** object (explained in earlier assignments).

11.4.5. Call methods from the **baseShape** to calculate the base shape's circumference and area.

12. The Cylinder class (Cylinder.cs)

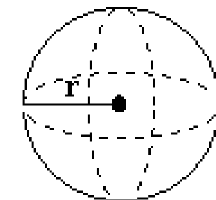


12.1. A cylinder occurs in many forms but here we consider only those which have a circular or elliptical base shape as shown in the figure. It is like a prism but has an elliptical base shape. This class inherits the class **Shape3D**.

12.2. Follow the steps as instructed for the prism and write the class in the same manner.

12.3. All the prism formulas for volume, **mantelArea** and the **totalArea** given above apply even for the Cylinder.

13. The Sphere class (Sphere.cs)



13.1. A sphere is a boll-shaped object. This class should also inherit the **Shape3D** class.

13.2. Use the same procedure to write this class using the following formulas:

```
volume = 4/3*Pi*radius3  = 4/3 * baseArea * radius
mantelArea = 4*Pi*radius2 = baseArea * 4
totalArea = mantelArea
```

13.3. To program radius3, you can use the following statement:

```
Math.Pow(radius, 3)
```

but it is better to use radius*radius*radius as it is faster and more exact.

13.4. The value of PI is also given in the Math class **Math.PI**.

© Farid Naisan

14. The FormMain class (FormMain.cs):

14.1. When the user presses the “Calculate” button, the type of the shape is determined and the necessary input is given. The shape type and the input need **not** to be remembered between the calculations. Therefore, the chosen shape object can be declared and created as a local variable in the method which starts the calculation and there is no need to declare a field variable of this type in the FormMain.

14.2. **Fields**: none.

15. Hints

15.1 To fill a ComboBox with values from an enum, you can use the structure Enum. The following example shows this and also how you can select an item in the combobox.:

```
cmbShape.Items.Clear();
cmbShape.Items.AddRange(Enum.GetNames(typeof(Shapes)));
cmbShape.SelectedIndex = (int)Shapes.Rectangle;
```

To map the selected item to an enum:item, you use type casting as illustrated in the code example below:

```
private Shapes GetFigure()
{
    return (Shapes)Enum.Parse(typeof(Shapes), cmbShape.Text, true);
}
```

15.2 You can use dynamic binding (or late binding) : To create an object of the selected type (Rectangle, Circle, etc), you can proceed as shown in this code example here.

```
Shapes figure = GetFigure();
Shape shape = null;

switch (figure)
{
    case Shapes.Rectangle:
    {
            shape = new Rectangle(width, height);
            lblArea.Text = ((Rectangle)shape).Area.ToString("0.00");
            lblPerimeter.Text = ((Rectangle)shape).Circumference.ToString("0.00");

            break;                  Type casting
    }

    case Shapes.Ellipse:
    case Shapes.Circle:
```

15.3 Dynamic binding is actually a type of polymorphism, so let's keep that for the Advanced Course. You can use the structure illustrated in the code snippet at the right.

Note: Both Area and Circumference are codded as properties in these examples, but you can write them as ordinary methods.

```
Shapes figure = GetFigure();

switch (figure)
{
    case Shapes.Rectangle:
    {
            Rectangle shape = new Rectangle(width, height);
            lblArea.Text = shape.Area.ToString("0.00");
            lblPerimeter.Text = shape.Circumference.ToString("0.00");
            break;
    }

    case Shapes.Ellipse:
    case Shapes.Circle:
```

Good Luck!

Programming is fun. Never give up. Ask for help!

Farid Naisan
Course Responsible and Instructor