Assignment: Polymorphism - Shapes

Polymorphism is the third pillar of object-oriented programming (OOP) after encapsulation and inheritance. Polymorphism allows an interface to be expressed in different ways. Real-world examples are commonplace; for example, the taxonomy used in biology to classify organisms is polymorphic.

In this assignment, you will create a class hierarchy of visual shapes inheriting from a common shape class; each shape knows how to draw itself. Please submit only the completed program file (Program […].cs) to the drop box.

Overview

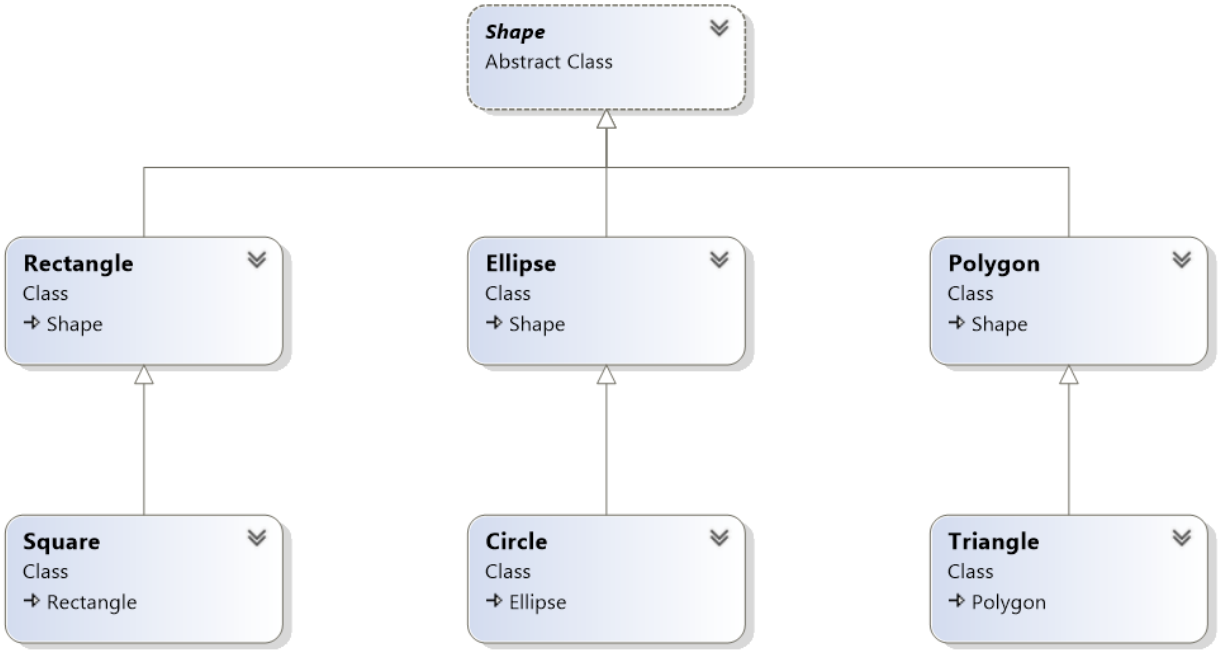
Polymorphism means "many forms." The form mammal, for example, includes the forms dog, cat, and human.

Here is an example of polymorphism in nature:

Polymorphism is really about type unification: derived types are instances of a common base type. The base type can define methods, abstract or virtual, that the derived type must override.

Polymorphism allows a derived type to be referenced through a base-type reference. Overridden methods in the derived type can then be invoked through the base-type reference.

In this assignment, you will create a class hierarchy of visual shapes—Rectangle, Square, Ellipse, Circle, and Polygon—all inheriting from an abstract Shape class that specifies an abstract Draw method.



Instructions

Setup

1. Download the assignment document (PDF) and the "Program […].cs" starter file.
2. Create a new C# Console application.
3. Delete the existing "Program.cs" file.
4. Add the "Program […].cs" file provided with the assignment.
5. Enter all code into the "Program […].cs" file.
6. Submit only the completed program file (Program […].cs) to the drop box.

Steps to Follow

Section 1: Defining the Shapes

1. Add references to the "System.Windows.Forms" and "System.Drawing" assemblies to the project. Import the System.Windows.Forms and System.Drawing namespaces.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.Drawing;
```

2. Define an abstract class named Shape having one abstract method, Draw, which takes a Graphics object parameter.

```
abstract class Shape
{
    abstract public void Draw(Graphics g);
    abstract public void Fill(Graphics g);
}
```

3. Define a Rectangle class that inherits from the Shape class.

```
abstract class Shape
{
    abstract public void Draw(Graphics g);
    abstract public void Fill(Graphics g);
}

class Rectangle : Shape
{
```

4. Insert a parameterized constructor that takes five parameters, caching them in private variables:
   - p: pen that determines the color, width, and style
   - b: brush that determines the color, width, and style
   - x: x-coordinate of upper-left corner of bounding rectangle
   - y: y-coordinate of upper-left corner of bounding rectangle
   - w: width of bounding rectangle
   - h: height of bounding rectangle

```
class Rectangle : Shape
{
    private Pen p;
    private Brush b;
    private float x, y, w, h;

    public Rectangle(Pen p, Brush b, float x, float y, float w, float h)
    {
        this.p = p;
        this.b = b;
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }

    public override void Draw(Graphics g)...
    public override void Fill(Graphics g)...
}
```

5. Implement the Draw method by delegating to the DrawRectangle method of the Graphics object.

```
public override void Draw(Graphics g)
{
    g.DrawRectangle(p, x, y, w, h);
}
```

6. Implement the Fill method by delegating to the FillRectangle method of the Graphics object.

```
public override void Fill(Graphics g)
{
    g.FillRectangle(b, x, y, w, h);
}
```

7. Define a Square class that inherits from the Rectangle class. Provide a constructor that takes five parameters; forward the constructor call to the base-class constructor.

```
class Square : Rectangle
{
    public Square(Pen p, Brush b, float x, float y, float s) :
        base(p, b, x, y, s, s) {}
}
```

8. Define an Ellipse class that inherits from the Shape class. Implement the Draw and Fill methods.

```
class Ellipse : Shape
{
    private Pen p;
    private Brush b;
    private float x, y, w, h;

    public Ellipse(Pen p, Brush b, float x, float y, float w, float h)...

    public override void Draw(Graphics g)...
    public override void Fill(Graphics g)...
}
```

9. Define a Circle class that inherits from the Ellipse class. Provide a constructor that takes five parameters; forward the constructor call to the base-class constructor.

```
class Circle : Ellipse
{
    public Circle(Pen p, Brush b, float x, float y, float r) :
        base(p, b, x, y, 2 * r, 2 * r) {}
}
```

10. Define a Polygon class that inherits from the Shape class. Provide a constructor that takes a Pen, a Brush, and an array of Points, and caches these values. Implement the Draw and Fill methods.

```
class Polygon : Shape
{
    private Pen p;
    private Brush b;
    private Point[] points;

    public Polygon(Pen p, Brush b, Point[] points)
    {
        this.p = p;
        this.b = b;
        this.points = points;
    }

    public override void Draw(Graphics g)...
    public override void Fill(Graphics g)...
}
```

11. Define a Triangle class that inherits from the Polygon class. Provide a constructor that takes five parameters; forward the constructor call to the base-class constructor.

```
class Triangle : Polygon
{
    public Triangle(Pen p, Brush b, Point x, Point y, Point z) :
        base(p, b, new Point[] { x, y, z }) { }
}
```

12. Build the project to make sure everything works.

Section 2: Polymorphism on Display

1. Add the following Form class to the same code module.

```
class ShapeForm : Form
{
    public ShapeForm()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        this.SuspendLayout();
        this.ClientSize = new System.Drawing.Size(600, 450);
        this.Name = "ShapeForm";
        this.Text = "Polymorphism: The Shape Form";
        this.Paint += new System.Windows.Forms.PaintEventHandler(this.ShapeForm_Paint);
        this.ResumeLayout(false);
    }

    private void ShapeForm_Paint(object sender, PaintEventArgs e)
    {
    }

    static void Main(string[] args)
    {
        Application.Run(new ShapeForm());
    }
}
```

2. Declare a private field to hold a list of shapes.

```
class ShapeForm : Form
{
    private List<Shape> shapes = new List<Shape>();
    public ShapeForm()
    {
        InitializeComponent();
    }
```

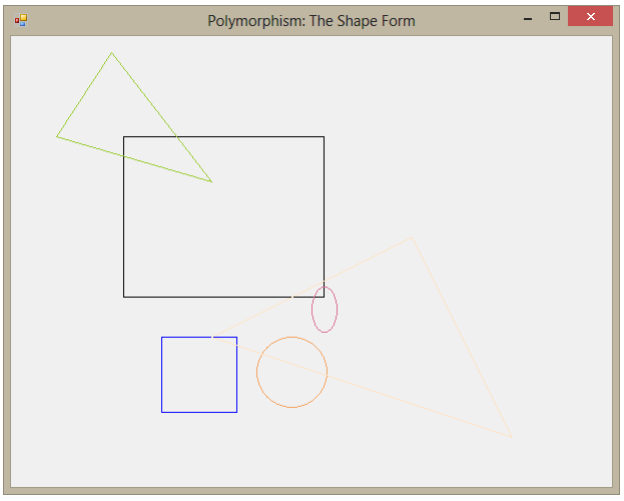
3. Create some shapes inside the ShapeForm constructor. Here are some samples.

```
class ShapeForm : Form
{
    private List<Shape> shapes = new List<Shape>();
    public ShapeForm()
    {
        InitializeComponent();

        // Add some shapes here
        shapes.Add(new Rectangle(Pens.Black, Brushes.Black, 112, 100, 200, 160));
        shapes.Add(new Square(Pens.Blue, Brushes.Blue, 150, 300, 75));
        shapes.Add(new Ellipse(Pens.PaleVioletRed, Brushes.PaleVioletRed, 300, 250, 25, 45));
        shapes.Add(new Circle(Pens.SandyBrown, Brushes.SandyBrown, 245, 300, 35));
        shapes.Add(new Triangle(Pens.Bisque, Brushes.Bisque,
            new Point(200, 300), new Point(400, 200), new Point(500, 400)));
        shapes.Add(new Polygon(Pens.YellowGreen, Brushes.YellowGreen,
            new Point[] { new Point(45, 100), new Point(100, 16), new Point(200, 145) }));

    }
```

4. In the ShapeForm Paint event handler, enumerate the list of shapes, calling the Draw method on each shape. Note the use of classic polymorphism here. Everything is just a shape. The form knows nothing about any specific shape.

```
private void ShapeForm_Paint(object sender, PaintEventArgs e)
{
    foreach (Shape shape in shapes)
    {
        shape.Draw(e.Graphics);
    }
}
```

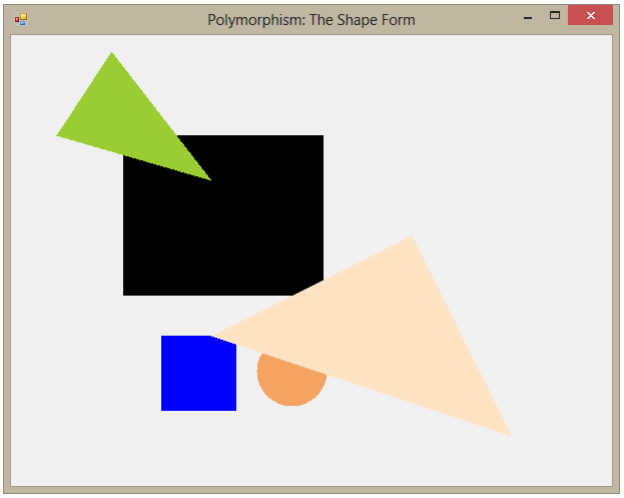
5. Run the application. Minimize the form; maximize it; resize it. The form repaints itself each time by calling the paint event.



6. In the ShapeForm_Paint event handler, enumerate the list of shapes, calling the Fill method on each shape.

```
private void ShapeForm_Paint(object sender, PaintEventArgs e)
{
    foreach (Shape shape in shapes)
    {
        shape.Fill(e.Graphics);
    }
}
```

7. Run the application. Minimize the form; maximize it; resize it. The form repaints itself each time by calling the paint event.



Summary

This assignment was a celebration of polymorphism. Polymorphism makes inheritance useful. Derived types supporting a common interface may provide a custom implementation for an interface method. A base-type reference to a derived-type instance will call the custom implementation when this method is invoked.

Polymorphism allows for generalization; groups of related objects are handled uniformly. The natural consequence is a cleaner and simpler object model and increased opportunities for code reuse.