School of Science, Computing and Engineering Technologies

Object Oriented Programming

Pass Task 4.1: Drawing Program — Multiple Shape Kinds

Overview

In this task, you extend the **ShapeDrawer** application that you created in "*3.3P Drawing Program – A Drawing Class*."

Purpose:	Learn to apply and use inheritance to create a family of related classes and objects, and experiment with polymorphism, a mechanism to interact with a group of related objects in an uniform way.
Task:	Extend the shape drawing program to allow for different types of shapes to be drawn on the screen.
Deadline:	Due by the start of week six, Monday, 8 April 2024.

Submission Details

All students have access to the Adobe Acrobat tools. Please print your solution to PDF and combine it with the screenshots taken for this task.

- C# code files of the classes created.
- Screenshot of running program (i.e., *SplashKit* window).

Instructions

Continue with the **ShapeDrawer** application that you developed in task *3.3P Drawing Program – A Drawing Class*. You may want to archive your previous work before extending the application.

Currently, your program enables you to draw many shapes of the same type (i.e., rectangle). In this task, you extend your program to allow for many different types of shapes to be drawn (e.g., rectangles, circles, and lines). This requires additional program logic and techniques. In particular, you will use *inheritance* to create a *family of related classes* and use *method overriding* to allow *polymorphism* in your application. Polymorphism is feature in object-oriented programming languages that permits a specific member function to use varying types of variables at different times.

The following UML class diagram illustrates the inheritance relationship that you are going to implement in this task. There are four classes: *Shape*, *MyRectangle*, *MyCircle*, and *MyLine*. Inheritance defines a "*is-a*" relationship between classes as well as objects (e.g., class *MyRectangle* is a kind of class *Shape*, *MyRectangle* objects are a kind of *Shape* objects). In UML, an inheritance relation is captured by drawing a line between the *base class* and the *derived class* with a hollow triangle pointing at the base class. So, in the UML diagram below, class *Shape* is that base class and the classes *MyRectangle*, *MyCircle*, and *MyLine* are the derived classes. (Please note that the base class is sometimes called *parent class* or *super class*, and the derived class is sometimes called *inherited class* or *subclass*.)

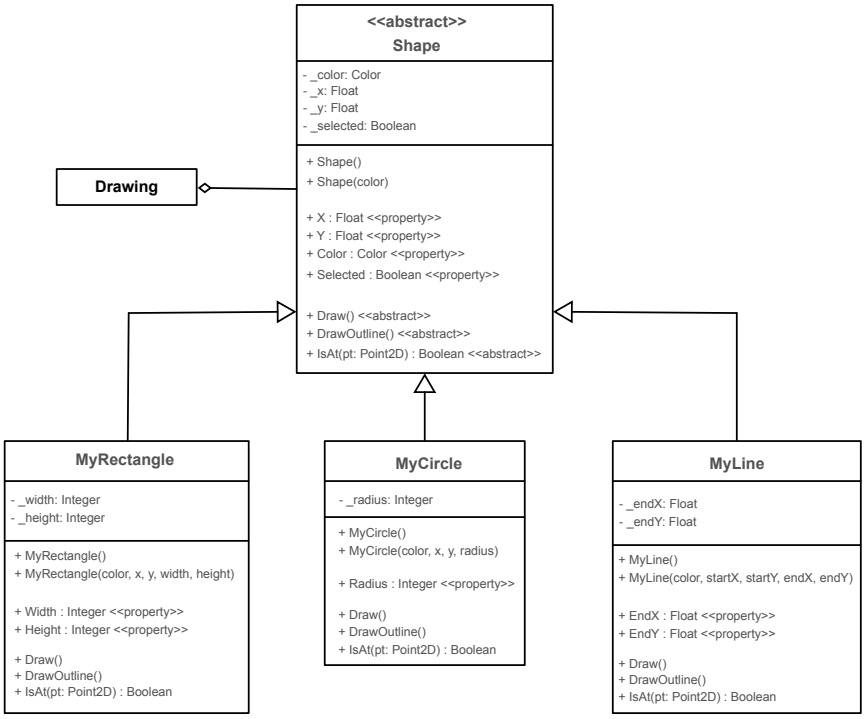


Figure 1: Shape Hierarchy.

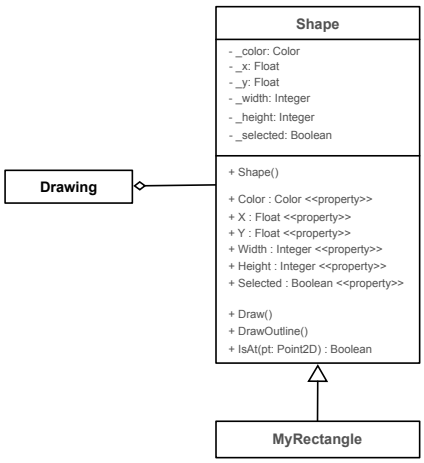
Please note that class *Shape* has been annotated with <<abstract>>. That is, you cannot create objects of class *Shape* any longer. In addition, the methods ***Draw***, ***DrawOutline***, and ***IsAt*** are all abstract now as well. This means, class *Shape* does not provide an implementation for those methods. They have to be implemented (or *overridden*) in the derived classes.

Note: Inheritance and polymorphism can be challenging concepts at first. Completing this task is the first step towards mastering them.

1. Open your **ShapeDrawing** solution.
2. Create a new class called *MyRectangle*.
3. Make class Shape the base class of class MyRectangle.

```
public class MyRectangle : Shape
{
}
```

By making class *Shape* the base class of class *MyRectangle* allows class *MyRectangle* to inherit from class *Shape*. Class *MyRectangle* is a kind of *Shape* that knows its location, its size, its color, and whether it is selected. In addition, you can tell a *MyRectangle* object to ***Draw***, to ***DrawOutline***, and ***IsAt***. The following UML fragment summarizes these facts.



Inheritance is primary mechanism for code reuse in object-oriented programming languages. Class *MyRectangle* is a fully functional class, due to inheriting and reusing all the code that you have defined in class *Shape*. Hence, you can use objects of class *MyRectangle* where objects of class *Shape* are expected.

4. Go to the ***Main*** function in *Program.cs*.
5. Locate the ***if***-statement that checks whether the user has clicked the left mouse button. Substitute ***new*** *Shape*() with ***new*** *MyRectangle*() as shown below:

```
if (SplashKit.MouseClicked(MouseButton.LeftButton))
{
    Shape newShape = new MyRectangle();

    newShape.X = SplashKit.MouseX();
    newShape.Y = SplashKit.MouseY();

    myDrawing.AddShape(newShape);
}
```

6. Run the program and add some rectangles (drawn at the current mouse pointer position).

   Your program works. It uses *subtype polymorphism* (sometimes also called inclusion polymorphism), which allows objects of a subclass to be used in a context where an object of the base class is expected. Here, ***new*** *MyRectangle*() creates ab object of class *MyRectangle,* which is a subclass of *Shape*. You can assign this object to ***newShape***, an object variable of class *Shape*. It works, because objects of class *MyRectangle* support all public methods of class *Shape*. This also means, we can safely pass objects of class *MyRectangle* to the method ***AddShape*** even though method ***AddShape*** takes as parameter an object of class *Shape*.

Note: Polymorphism means "*many forms*". It is one of the key features in object-oriented programming. There are four types of polymorphism (see the "On understanding types, data abstraction, and polymorphism" by Peter Wegner and Luca Cardelli for detailed analysis):

- Coercion:

  ```
  float x = 10;
  ```

  The integer literal 10 is converted (by the compiler) to a floating point value.

- Overloading:

  `FillRect(clr, x, y, w, h);` and `FillRect(clr, myRect);`

  There are two versions of method ***FillRect***. Method ***FillRect*** is overloaded. The compiler uses the actual arguments passed to ***FillRect*** to select the best matching method.

- Subtype:

  `Shape myShape = new MyRectangle();` or

  `myDrawing.AddShape(new MyRectangle();`

  Objects of a subclass can be used in a context where an object of the base class is expected. This includes assignments, parameters, and results of functions.

- Universal:

  `List<Shape>` or `List<int>`

  Class *List* is parameterized over the type of values it can maintain. Universal polymorphism provides the means to define *generic data types* in C#. Objects of class *List* provide the same features, irrespective whether it is a *List of Shapes* or a *List of ints*.

7. Create a new class called *MyCircle*, which is a subclass of *Shape*.

8. In *Program.cs*, make the following changes to class Program:

   8.1. Define a ***private*** enumeration called *ShapeKind*. This enumeration is used to determine what shape the user wants to create.

```
public class Program
{
    private enum ShapeKind
    {
        Rectangle,
        Circle
    }

    public static void Main()
    {
```

> **Note**: In C#, you can declare types within other types. This type is encapsulated within its enclosing class. This can be a useful feature for types like ***enum*** *ShapeKind*, which should only be visible within class *Program*.

   8.2. Create a *ShapeKind* variable in function ***Main*** called ***kindToAdd***.

   8.3. Initialize ***kindToAdd*** with *ShapeKind.Circle*.

   8.4. Inside the event loop (after *SplashKit.**ClearScreen***):

- Add an ***if***-statement to test if the user has typed the ***R***-key (*KeyCode.RKey*). If this is true, set ***kindToAdd*** to *ShapeKind.Rectangle*.
- Add an ***if***-statement to test if the user has typed the ***C***-key (*KeyCode.CKey*). If this is true, set ***kindToAdd*** to *ShapeKind.Rectangle*.
- Update your code to allow for different shapes to be created. Use the following code as a guide.

```
if (SplashKit.MouseClicked(MouseButton.LeftButton))
{
    Shape newShape;

    switch(kindToAdd)
    {
        case ShapeKind.Circle:
            newShape = new MyCircle();
            newShape.X = SplashKit.MouseX();
            newShape.Y = SplashKit.MouseY();
            break;

        default:
            newShape = new MyRectangle();
            newShape.X = SplashKit.MouseX();
            newShape.Y = SplashKit.MouseY();
            break;
    }

    myDrawing.AddShape(newShape);
}
```

> **Note**: Polymorphism allows you to assign objects of class *MyCircle* and *MyRectangle* to ***myShape***.

9. The above contains code duplication to position the shape. This is unnecessary. All *Shape* objects know their location. Fix the code duplication.
10. Compile and run the program.

Unfortunately, all shapes are draws the same way – as rectangles. You have used inheritance to create different classes for *MyRectangle* and *MyCircle*. But their behavior is still identical. We need to *override* aspects of the behavior in class MyRectangle and *MyCircle* to allow these classes to represent different shapes. This will also require some updates in class *Shape*. Start with class *MyCircle* first.

11. Add a ***_radius*** field that stores an integer value, and a ***Radius*** property to allow others to get and set the value of ***_radius***.
12. Add a *constructor* and set ***_radius*** to 50.
13. Switch to class *Shape*. Decorate the methods ***Draw*** and ***DrawOutline*** as ***virtual***. The ***Draw*** method is shown below.

```
public virtual void Draw()
{
    if (Selected)
    {
        DrawOutline();
    }

    SplashKit.FillRectangle(_color, _x, _y, _width, _height);
}
```

> **Note**: Declaring a method as virtual in C# allows subclasses to override the method. In addition, virtual methods are subject to *dynamic dispatch*, a process of selecting which implementation of a overridden (polymorphic) method to call at run time.

14. Return to class *MyCircle* and define an overridden version of method ***Draw***. Use the following code as a guide.

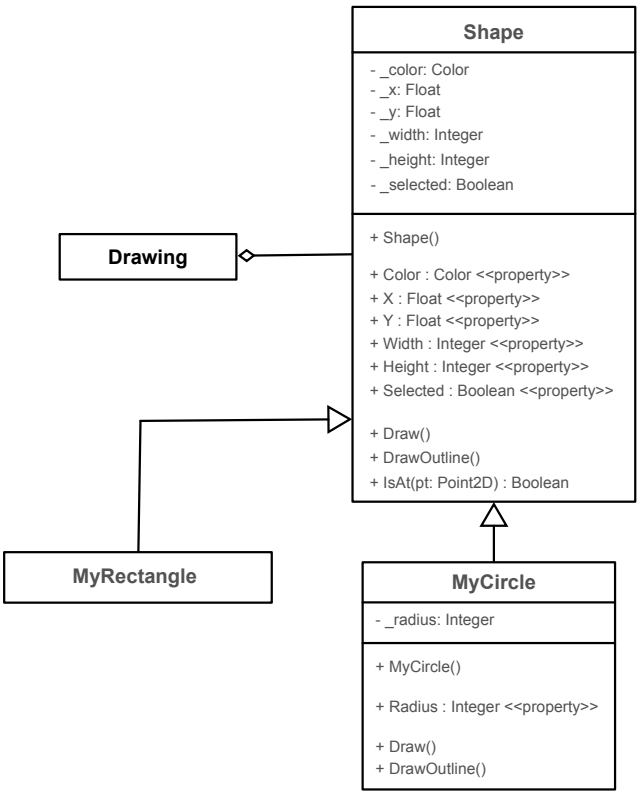
```
public override void Draw()
{
    if (Selected)
    {
        DrawOutline();
    }

    SplashKit.FillCircle(Color, X, Y, _radius);
}
```

Tip: Visual Studio can help you when overriding methods. Just start typing override and the Visual Studio will give you a list of the methods you can override. By default it calls **base**.Draw() which calls the method from class *Shape*.

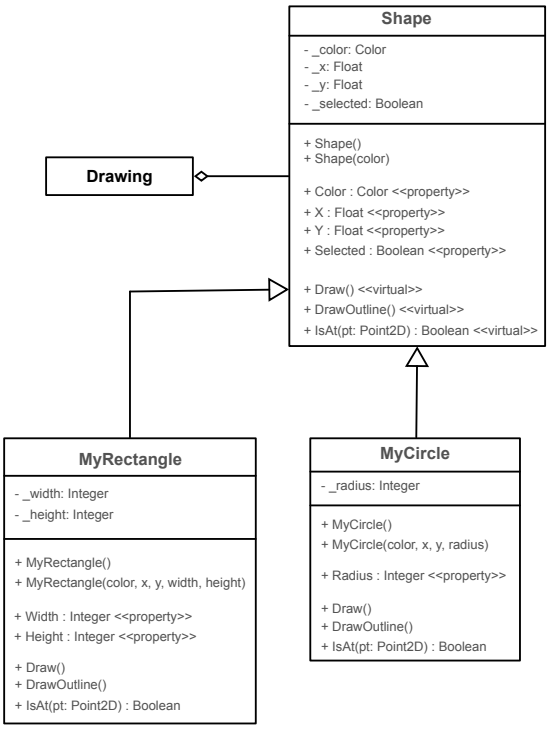
Note: The fields in the base class are private, so you need to use the public properties ***Color***, ***X***, and ***Y*** to access the circle's color and position.

15. Run the program. Circles and Rectangles are drawn correctly now.
16. In class *MyCircle*, override method ***DrawOutline*** and change it so that it draws a black circle with a radius 2 pixels larger than the circle's radius.
17. Run the program again, and try selecting and deleting circles and rectangles.

You can create circles and rectangles. But not everything works, especially selecting and deleting circles. The main reason for this is that you have not defined a circle-specific method ***IsAt***. In addition, our current code base does not reflect the design shown in Figure 1. You have only implemented the following UML diagram.



18. Rework your classes *Shape*, *MyRectangle*, and *MyCircle* so that they match the following UML diagram:



- For class *Shape*:
  - Move the ***_width*** and ***_height*** fields and the associated properties, ***Width*** and ***Height***, to class *MyRectangle*.
  - Add an overloaded *Shape* constructor that takes ***color*** as argument. It initializes the fields ***_x*** and ***_y*** with 0.0f and sets ***_color*** to ***color***.
  - Change the default constructor. You use a ***this****-call* to another defined constructor to initialize the object with *Color.Yellow*.
  - Change methods ***Draw*** and ***DrawOutline*** to have an empty method body.
  - Change method ***IsAt*** to be ***virtual*** and to return ***False***.
- For class *MyRectangle*:
  - Define a constructor that accepts the ***color***, the location (***x***, ***y***), and the size (***width***, ***height***) of the rectangle. This constructor must use a ***base***-call to initialize the color in the base class *Shape*. Use the following code as a guide. You need to add a using declaration for *SplashKitSDK* so that type *Color* is defined.

```
public MyRectangle(Color color, float x, float y, int width, int height) : base(color)
{
    X = x;
    Y = y;
    Width = width;
    Height = height;
}
```

  - Define a default constructor. You use a ***this***-*call* to another defined constructor to initialize the object with *Color.Green* for ***color***, 0.0f for ***x*** and ***y***, and 100 for ***width*** and ***height***.
  - Override the methods ***Draw***, ***DrawOutline***, and ***IsAt***. Use corresponding *Shape* methods defined in task *3.3P Drawing Program – A Drawing Class* as a guide.

- For class *MyCircle*:
  - Add an overloaded constructor that accepts a ***color*** and a ***radius***. This constructor must use a ***base***-call to initialize the color in the base class *Shape*.
  - Change the default constructor. You use a ***this***-*call* to another defined constructor to initialize the object with *Color.Blue* for ***color***, 50 for ***radius***.
  - Override method ***IsAt*** which checks if the point ***pt*** is within the area of the circle.

> **Tip**: If the distance between the center of the circle and a given point does not exceed the radius of the circle, then the point is in the circle.

19. Compile and run the program. Check that you can add, select, and delete both circles and rectangles.

> **Hint**: *SplashKit* offers some helpful methods for implementing ***IsAt***. Checkout *PointInCircle* and *CircleAt* for class *MyCircle*, and *PointOnLine* and *LineFrom* for class *MyLine*.

Everything works as expected. However, if you check the methods ***Draw***, ***DrawOutline***, and ***IsAt*** for class *MyRectangle* and *MyCircle*, then your notice that you have defined an overridden version for all and none of these overridden versions depends or uses the inherited code.

This is a common pattern in object-oriented design. You require all objects in a given class hierarchy to have a specific set of methods, so you introduce these methods in the root class of the class hierarchy. But only the subclasses define the actual behavior. The solution to this problem is that you declare the methods in the root class ***abstract***. Abstract member functions have no implementation. Derived classes must implement abstract methods via method overriding. In C#, a class that has abstract methods must be declared abstract also. You cannot create objects for abstract classes.

Note: Defining ***Draw***, ***DrawOutline***, and ***IsAt*** abstract promises that all shapes respond to these methods, but it is the derived classes of class *Shape* that fulfill this promise by implementing the methods.

20. Decorate class *Shape* as ***abstract***.

```
public abstract class Shape
{
    …
```

Note: The word "abstract" occurs often in programming language terminology and can relate to distinct concepts. For example, an abstract class cannot be instantiated, that is, you cannot create objects for an abstract class. Abstract methods have no implementation. They are part of the interface derived classes have to implement. Abstraction (or data abstraction) is the process of generalizing concrete details. Abstraction allows for the separation of usage (interface) from representation (implementation) of data.

21. Make method *Draw* in class Shape ***abstract***.

```
public abstract void Draw();
```

Note: Abstract methods are implicitly virtual. A derived class must override it.

22. Build and run the program. It should work as before.

Note: You program can no longer contain ***new*** *Shape*() or ***new*** *Shape*(***color***). Class *Shape* is abstract and you cannot create objects of class *Shape*.

23. Make methods ***DrawOutline*** and ***IsAt*** in class *Shape* abstract also. (Method ***IsAt*** in class *Shape* is no longer required to return a Boolean value.)
24. Build and run the program again. It works.

25. Add a new class, named *MyLine,* that implements the necessary features to be a kind of *Shape*. Refer to Figure 1 and classes *MyRectangle* and *MyCircle* for guidance.

> **Note**: By default, objects of class *MyLine* are drawn in red. In addition, you only have access to the mouse pointer position for the start of the line. For the endpoint, use hard-coded values of your choosing.

> **Hint**: To create the outline of a *MyLine* object, draw small circles around the start and end points of the line.

26. Adjust Program so that you can add lines after you have typed the L key.
27. Run the program. You should be able to add circles, rectangles, and lines, select them, and removing them.

Once your program is complete you can prepare it for your portfolio. This can be placed in your portfolio as evidence of what you have learnt.

1. Review your code and ensure it is formatted correctly.
2. Run the program and use your preferred screenshot program to take a screenshot of the Terminal showing the program's output.
3. Save and backup your work to multiple locations, if possible.
    - Once you your program is working you do not want to lose your work.
    - Work on your computer's storage device most of the time, but backup your work when you finish each task.
    - You may use a cloud storage provider to safely store your work.

USB and portable hard drives are good secondary backups, but there is a risk that the drive gets damaged or lost.

Once your program is working correctly you can prepare it for your portfolio.

Add a screenshot of the program working, and your source code.

Assessment Criteria

Make sure that your task has the following in your submission:

- The "Universal Task Requirements" (see Canvas) have been met.
- Your program (as text or screenshot).
- Screenshots of running program (i.e., *SplashKit* window) that show the different aspects of the application: adding shapes, selecting shapes, and removing shapes.