School of Science, Computing and Engineering Technologies

Object Oriented Programming

Pass Task 3.3: Drawing Program - A Drawing Class with your own attributes

Overview

In this task, you extend the **ShapeDrawer** application that you created in "*2.3P Drawing Basic Shapes with your own attributes*."

Purpose:	Learn to apply object-oriented programming techniques related to collaboration and the use of framework classes.
Task:	Extend the shape drawing program to allow for many shapes to be drawn on the screen. **The task contains personalized requirements.**
Deadline:	Due by the end of week four, **Friday, 30 May 2025, 23:59 Hanoi Time** (**Firmed**).

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).

SWINBURNE UNIVERSITY OF TECHNOLOGY

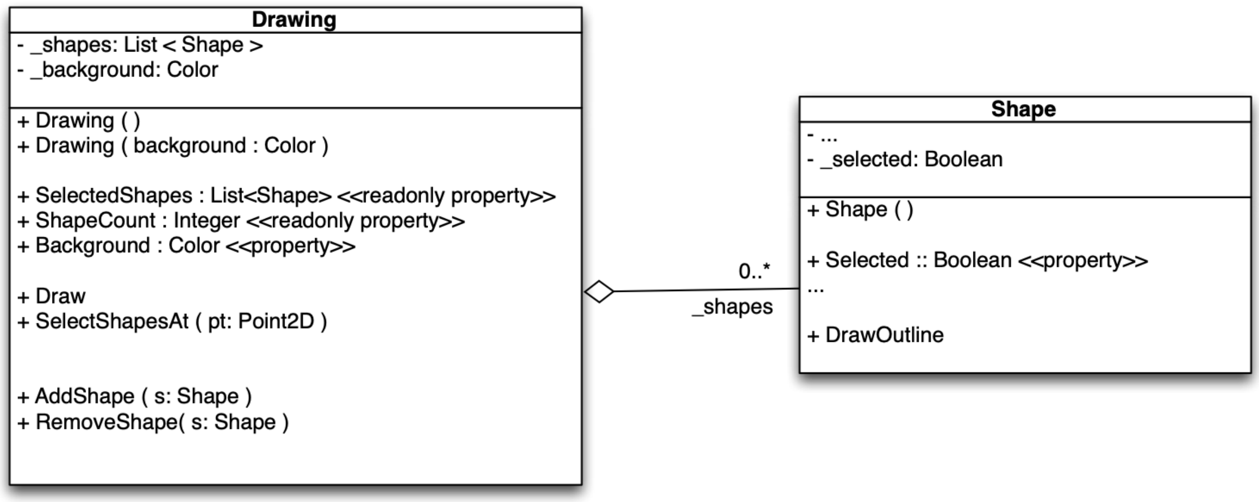
Instructions

Continue with the **ShapeDrawer** application that you developed in task *2.3P Drawing Program – A Basic Shape*. You may want to archive your previous work before extending the application.

Presently, your program only allows you to draw one shape (i.e., a rectangle at the current mouse pointer position). However, a general drawing program should be able to draw many shapes to the screen. To achieve this, extend your program with a new class, called *Drawing*. Class *Drawing* defines a container of *Shape* objects and provides mechanisms to *add shapes* to the collection, to *select shapes* for drawing, to *remove shapes* from the collection, and to *draw* the selected shapes to the screen. In addition, class *Drawing* supports a set of properties to query information about the collection or to alter drawing attributes.

Note: You can probably think of many other operations for the drawing, such as saving to file etc. This will be a useful part of the overall program.

The following UML diagram illustrates the new system design.


Drawing
- _shapes: List < Shape >
- _background: Color
+ Drawing ( )
+ Drawing (background : Color)
+ SelectedShapes : List<Shape> <<readonly property>>
+ ShapeCount : Integer <<readonly property>>
+ Background : Color <<property>>
+ Draw
+ SelectShapesAt (pt: Point2D)
+ AddShape (s: Shape)
+ RemoveShape(s: Shape)
0..*
_shapes
Shape
- ...
- _selected: Boolean
+ Shape ( )
+ Selected :: Boolean <<property>>
...
+ DrawOutline

The line between class *Drawing* and class *Shape*, with the hollow diamond, indicates an *aggregation* between class *Drawing* and class *Shape*. It states that objects of class *Drawing* have a "***_shapes***" collection of zero or more objects of class *Shape* as attribute.

Note: The ***_shapes*** field is actually a list of *Shape* objects, which can have zero of more elements.

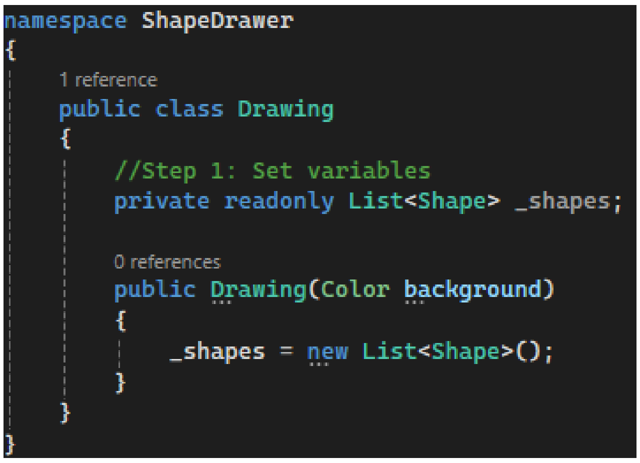
1. Open your **ShapeDrawer** solution.
2. Add a new class, called *Drawing*, to your application.
3. Add a private, read-only field ***_shapes***. Use *List*<*Shape*> as the type for ***_shapes***.

Note: Class *List* is a member of the *System.Collections.Generic* namespace. This namespace is implicitly available to every C# project. Hence, you do not need to add a ***using*** declaration for this namespace at the top or your C#-class file.

Tip: Mark fields as ***readonly*** if you are not going to change them after the object is created. In case of ***_shapes***, we will always be using the same *List* object, so we do not want to change the field.

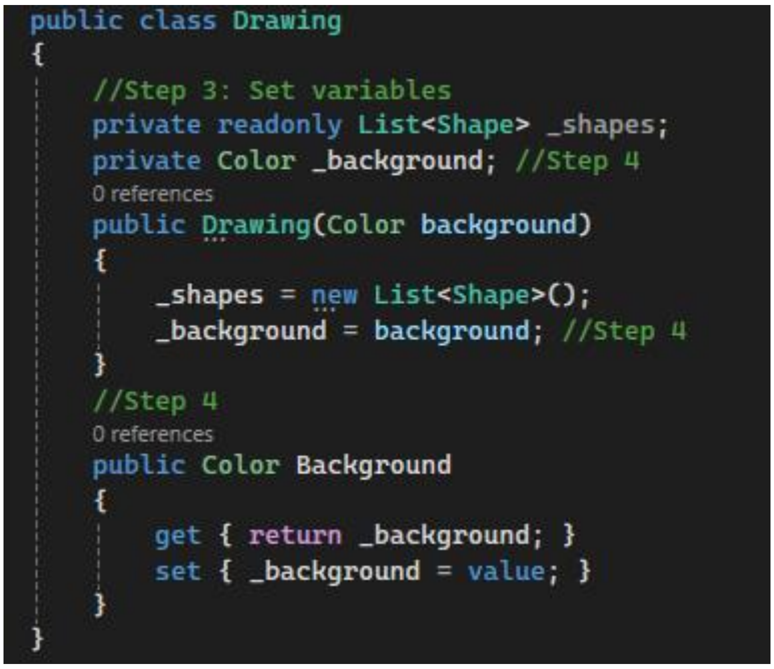
Note: A ***readonly*** field cannot be changed, meaning that you cannot assign a new value to the field. However, the object that field refers to can change, and will change in this case as we add and remove shapes from the list.

```
public class Drawing
{
    private readonly List<Shape> _shapes;
    …
    public Drawing(Color background)
    {
        _shapes = new List<Shape>();
        …
    }
}
```



```
namespace ShapeDrawer
{
    1 reference
    public class Drawing
    {
        //Step 1: Set variables
        private readonly List<Shape> _shapes;

        0 references
        public Drawing(Color background)
        {
            _shapes = new List<Shape>();
        }
    }
}
```

4. Add a private ***_background*** field and public ***Background*** property for the background color. Use type *Color* as type. You need to add ***using*** *SplashKitSDK*; at the top of your *Drawing.cs* file to make type *Color* available.
5. Create the constructor that takes in the ***background*** color as a parameter.
   - Create a new *List*<*Shape*> object and assign it to the ***_shapes*** field.
   - Initialize the ***_background*** to the supplied background color.



```
public class Drawing
{
    //Step 3: Set variables
    private readonly List<Shape> _shapes;
    private Color _background; //Step 4
    0 references
    public Drawing(Color background)
    {
        _shapes = new List<Shape>();
        _background = background; //Step 4
    }
    //Step 4
    0 references
    public Color Background
    {
        get { return _background; }
        set { _background = value; }
    }
}
```

Object-oriented programming languages come with a rich set of class libraries. These libraries include a set *collection classes*, like *List*, that manage *collections of objects*. Collection classes define the smarts needed to maintain a collections of objects for you. For example, the *List* class provides the intelligence to manage a *dynamic array* of objects. You can add, remove, and fetch objects from the list, etc. A *List* has everything you need if you want a dynamic array of some kind.

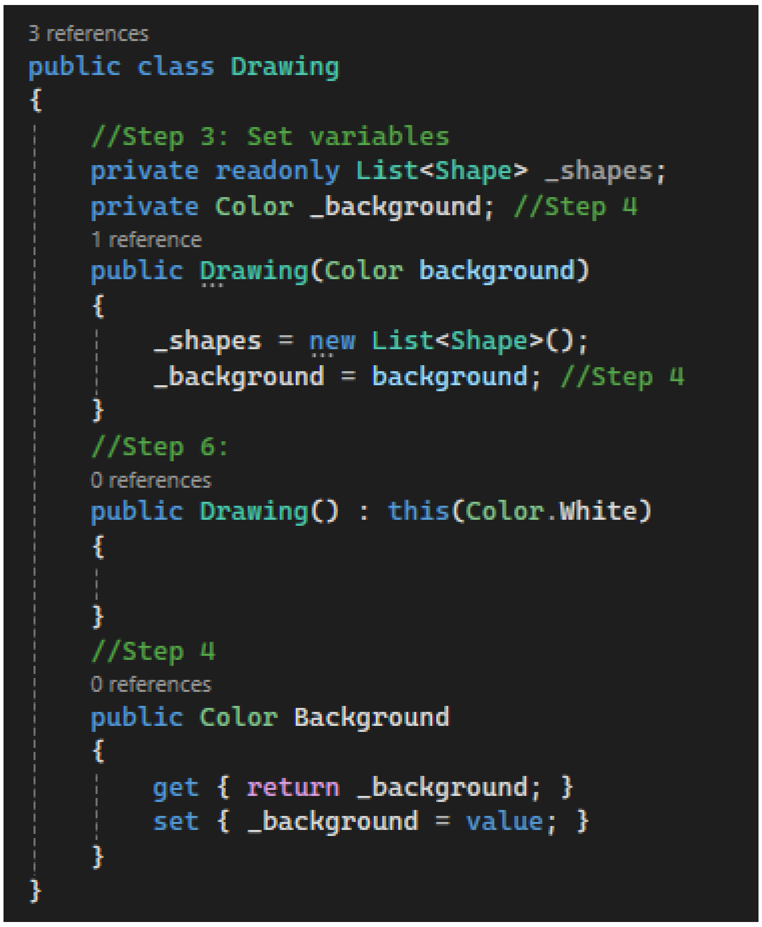
In C# and the .NET framework, collection classes are *generic datatypes*. Generic datatypes provide a uniform set of operations for the objects contained in the collection across all possible instantiations. For example, the specification *List*<*Shape*> instantiates the generic type *List* with class *Shape.* The result is a list of *Shape* objects. Generic collections "*remember*" which type of objects they store. So, if you can add a *Shape* object to a *List*<*Shape*> collection, then you get *Shape* objects back if you ask the *List*<Shape> collection to return an element.

6. Add a second constructor with no parameters to class **Drawing**. A constructor with no parameters is called *default constructor*. Default constructors initializes objects with predefined values.

   When defining the default constructor, you want to avoid code duplication. For this reason, you use a ***this***-*call* to another defined constructor to initialize the object. In case of class Drawing, use the constructor that takes a background color for the ***this***-call, using *Color.White* as argument.

```
public class Drawing
{
    public Drawing ( Color background ) { … }
    public Drawing ( ) : this ( Color.White )
    {
         // other steps could go here…
    }
}
```

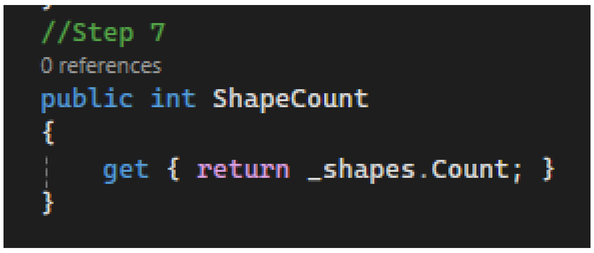
Note: When you use a ***this***-*call*, all member variables should be initialized in the target constructor. For example, calling ***new*** *Drawing*() must return an object that is equal to an object returned by ***new*** *Drawing*(*Color.White*).



```
3 references
public class Drawing
{
    //Step 3: Set variables
    private readonly List<Shape> _shapes;
    private Color _background; //Step 4
    1 reference
    public Drawing(Color background)
    {
        _shapes = new List<Shape>();
        _background = background; //Step 4
    }
    //Step 6:
    0 references
    public Drawing() : this(Color.White)
    {

    }
    //Step 4
    0 references
    public Color Background
    {
        get { return _background; }
        set { _background = value; }
    }
}
```

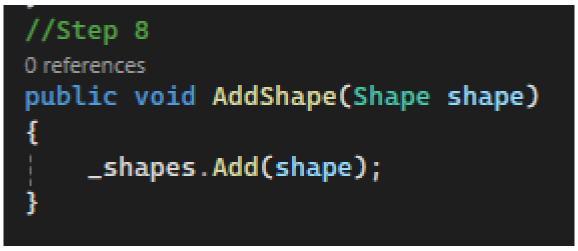
7. Add a read-only ***ShapeCount*** property to class *Drawing* that returns the ***Count*** from the ***_shapes*** list collection object.

Tip: Follow the *single responsibility principle* (SRP) in your object-oriented design. For example, which actor in your system is responsible for computing the number of objects in ***_shapes***? You do not reimplement this feature in class *Drawing*. The responsibility rests with class *List*. So, you ask List to compute its Count and return the result as value for read-only property ***ShapeCount***.



```
//Step 7
0 references
public int ShapeCount
{
    get { return _shapes.Count; }
}
```

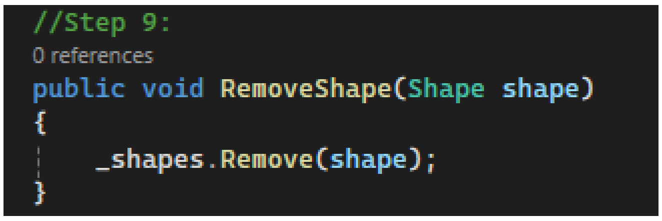
8. Create the ***AddShape*** method in class *Drawing* that adds the shape it receives to its list of shapes.



```
//Step 8
0 references
public void AddShape(Shape shape)
{
    _shapes.Add(shape);
}
```

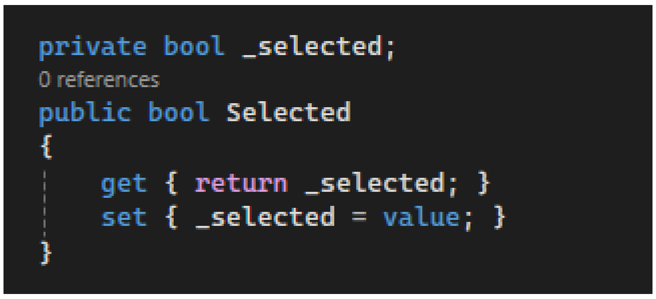
9. Create the ***RemoveShape*** method in class *Drawing* that removes the shape it receives from its list of shapes.

Note: List's ***Remove*** method returns ***true*** if it successfully removed an item. You may want to use a discard assignment "_ = …" to ignore the return value.



```
//Step 9:
0 references
public void RemoveShape(Shape shape)
{
    _shapes.Remove(shape);
}
```

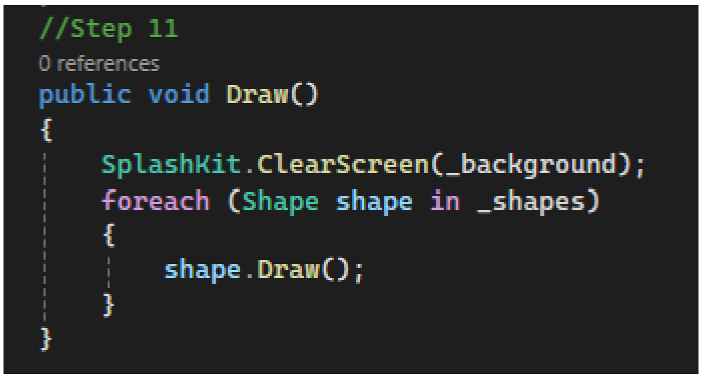
10. Switch to your *Shape* class and add a private ***_selected*** field and a public ***Selected*** property. (You do not need to change the constructor. A Boolean field is initialized to ***false*** by default.)



```
private bool _selected;
0 references
public bool Selected
{
    get { return _selected; }
    set { _selected = value; }
}
```

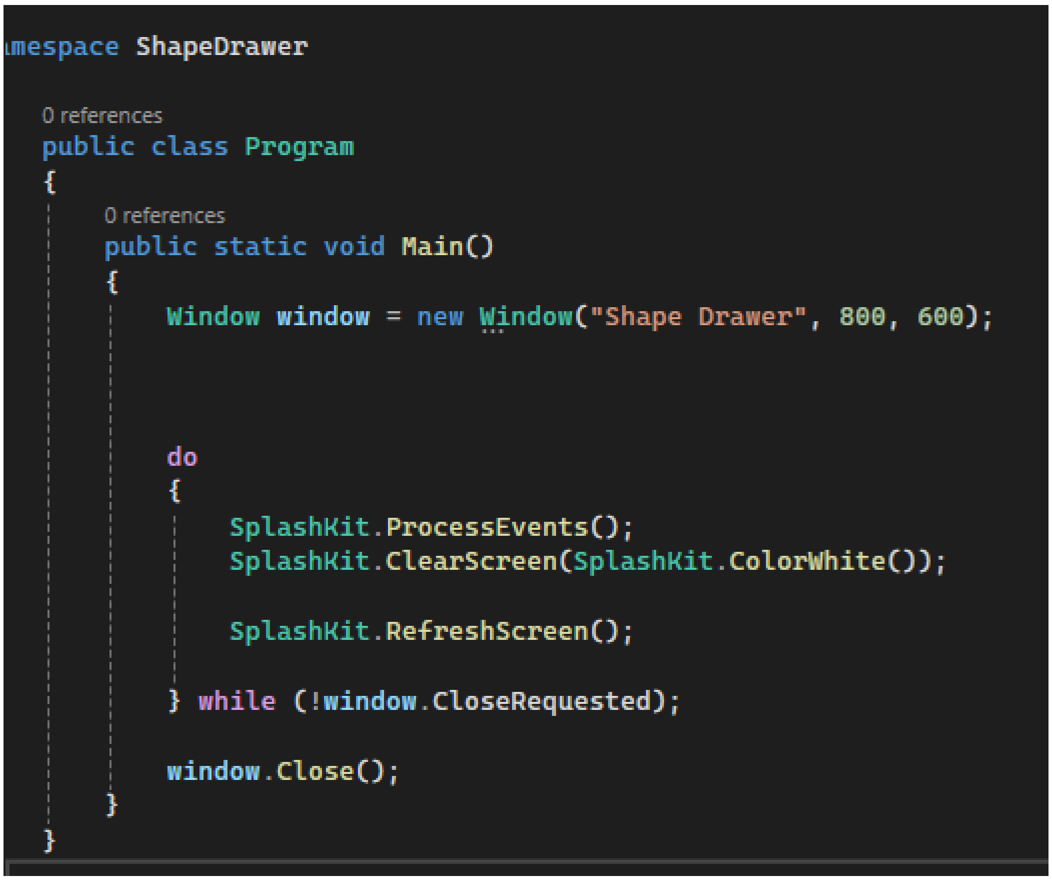
11. Return to class *Drawing* and add a ***Draw*** method. Tell *SplashKit* to ***ClearScreen*** using the ***_background*** color as argument and then loop over each shape and tell it to ***Draw*** itself.

Note: The Drawing class does not actually draw the shapes, it asks the shapes to draw themselves. This is the idea of collaboration in object-oriented programming.



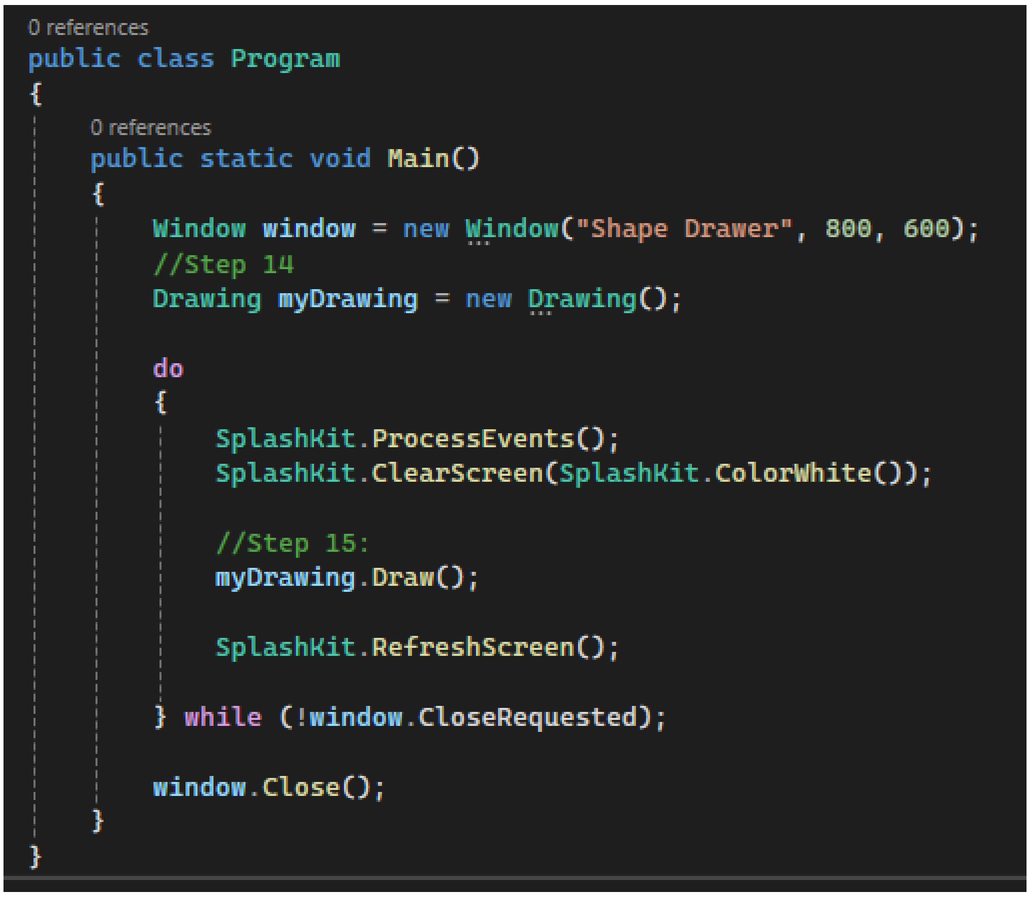
```
//Step 11
0 references
public void Draw()
{
    SplashKit.ClearScreen(_background);
    foreach (Shape shape in _shapes)
    {
        shape.Draw();
    }
}
```

12. Go to the ***Main*** function in *Program*.cs.
13. Remove the ***myShape*** variable and all code referring to ***myShape***.



```
mespace ShapeDrawer

    0 references
    public class Program
    {
        0 references
        public static void Main()
        {
            Window window = new Window("Shape Drawer", 800, 600);



            do
            {
                SplashKit.ProcessEvents();
                SplashKit.ClearScreen(SplashKit.ColorWhite());

                SplashKit.RefreshScreen();

            } while (!window.CloseRequested);

            window.Close();
        }
    }
```

14. Create a *Drawing* object ***myDrawing*** using the default constructor outside the ***do-while*** loop.



```
0 references
public class Program
{
    0 references
    public static void Main()
    {
        Window window = new Window("Shape Drawer", 800, 600);
        //Step 14
        Drawing myDrawing = new Drawing();

        do
        {
            SplashKit.ProcessEvents();
            SplashKit.ClearScreen(SplashKit.ColorWhite());

            //Step 15:
            myDrawing.Draw();

            SplashKit.RefreshScreen();

        } while (!window.CloseRequested);

        window.Close();
    }
}
```

15. Inside the ***do-while*** loop in ***Main*** (after *SplashKit.**ClearScreen***):

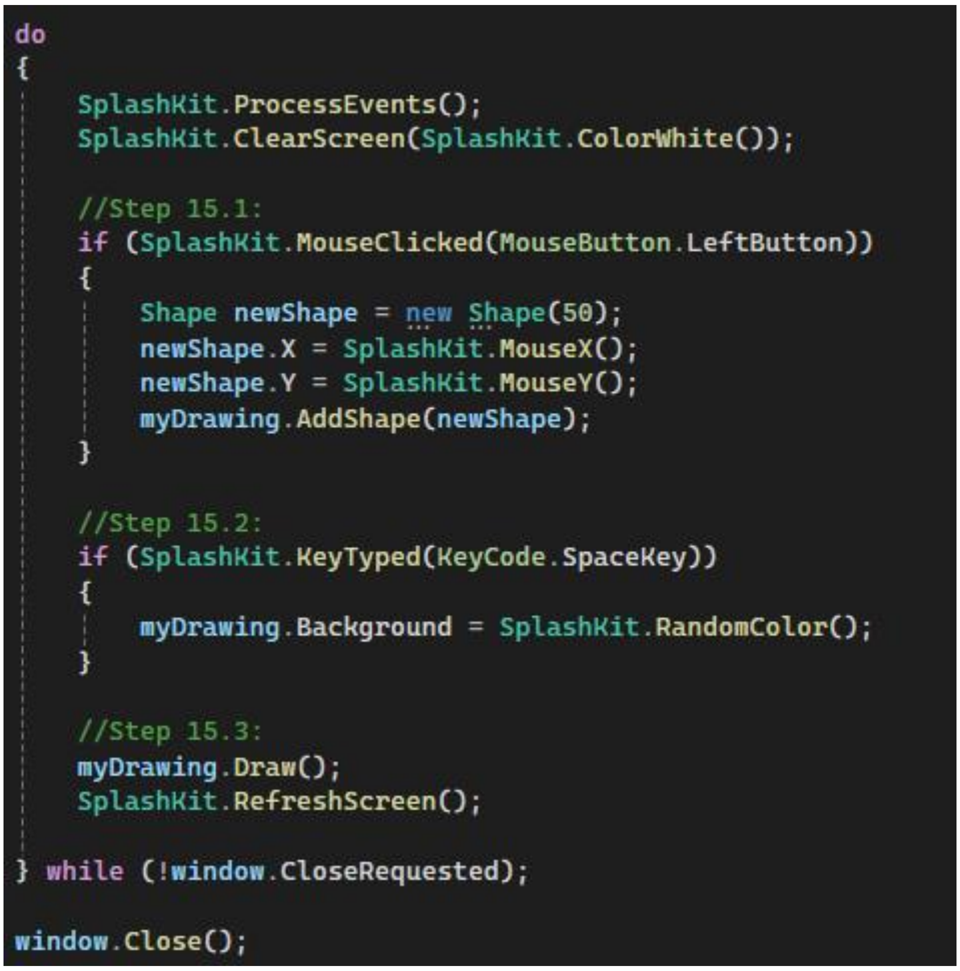
Note: The ***do-while*** loop in ***Main*** is also called *event loop*.

15.1. Check if the user has clicked the left mouse button. In this case, add a **new** *Shape* to ***myDrawing*** using the current mouse pointer location.

Hint: You should use a local variable to create a *Shape* object using the default constructor, then alter the ***X*** and ***Y*** location of the shape using the mouse pointer location, and finally add the newly created shape to ***myDrawing***.

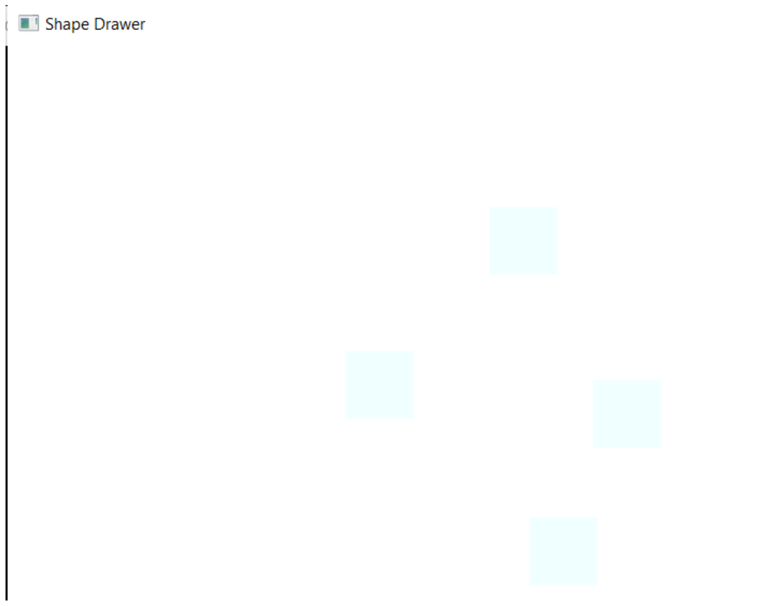
15.2. Change the background color of ***myDrawing*** to a new random color when the user presses the space bar.

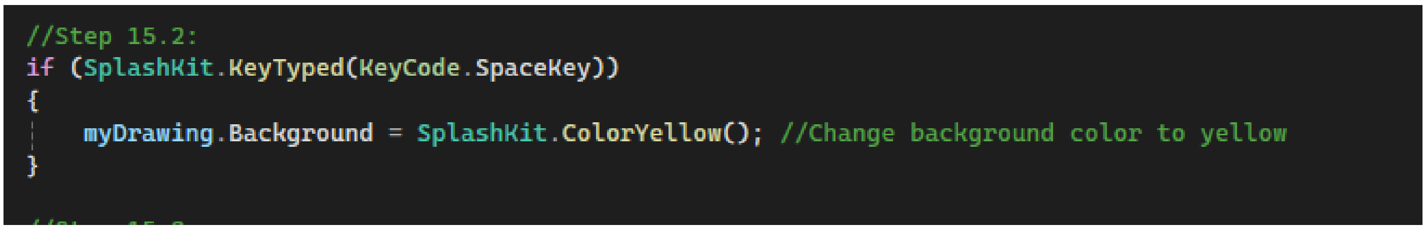
15.3. Tell ***myDrawing*** to ***Draw*** before *SplashKit.**RefreshScreen***.



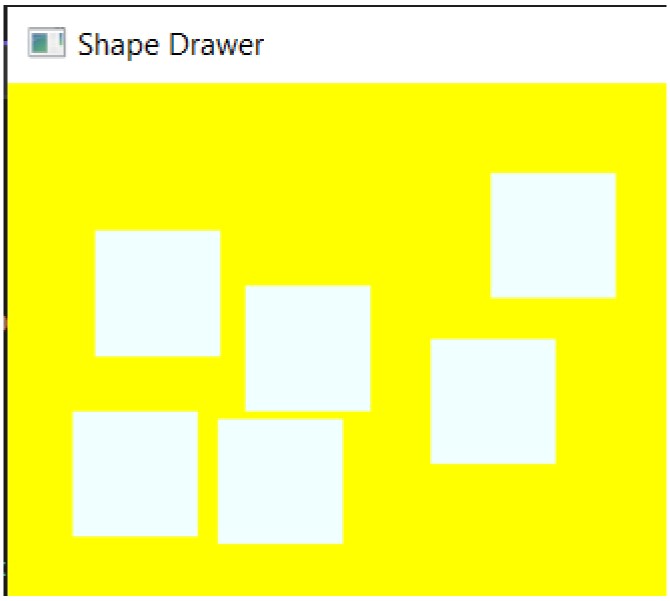
```
do
{
    SplashKit.ProcessEvents();
    SplashKit.ClearScreen(SplashKit.ColorWhite());

    //Step 15.1:
    if (SplashKit.MouseClicked(MouseButton.LeftButton))
    {
        Shape newShape = new Shape(50);
        newShape.X = SplashKit.MouseX();
        newShape.Y = SplashKit.MouseY();
        myDrawing.AddShape(newShape);
    }

    //Step 15.2:
    if (SplashKit.KeyTyped(KeyCode.SpaceKey))
    {
        myDrawing.Background = SplashKit.RandomColor();
    }

    //Step 15.3:
    myDrawing.Draw();
    SplashKit.RefreshScreen();

} while (!window.CloseRequested);

window.Close();
```

16. Compile and run your program. Add shapes and change the background color.





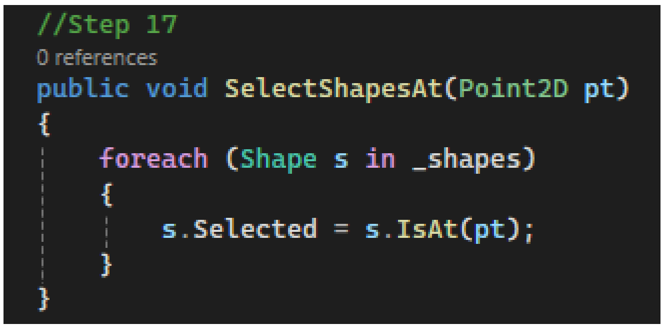
```
//Step 15.2:
if (SplashKit.KeyTyped(KeyCode.SpaceKey))
{
    myDrawing.Background = SplashKit.ColorYellow(); //Change background color to yellow
}
```



17. Switch to class *Drawing*. Add the ***SelectShapesAt*** method to your *Drawing* class. Use the following pseudocode as a guide.

```
SelectShapesAt(pt)
// parameter pt is a 2D point
foreach s in _shapes
    if Tell s to IsAt(pt)
        s.Selected := true
    else
        s.Selected := false
```

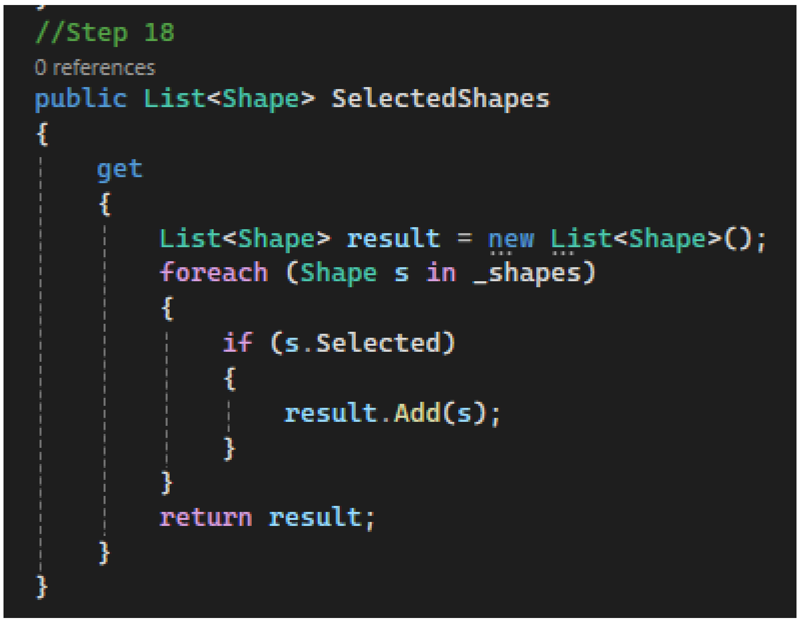
Hint: Avoid the ***if***-statement inside the ***foreach***-loop. The result of ***IsAt*** is a Boolean.



```
//Step 17
0 references
public void SelectShapesAt(Point2D pt)
{
    foreach (Shape s in _shapes)
    {
        s.Selected = s.IsAt(pt);
    }
}
```

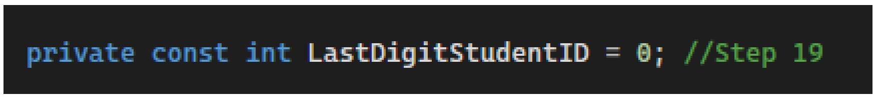
18. Use the following pseudocode to implement the read-only property ***SelectedShapes***.

```
    SelectedShapes:get()
1:  Let result by a new List of Shape objects.
2:  foreach s in _shapes
3:        if s.Selected
4:              Tell result to Add(s)
5:  return result
```

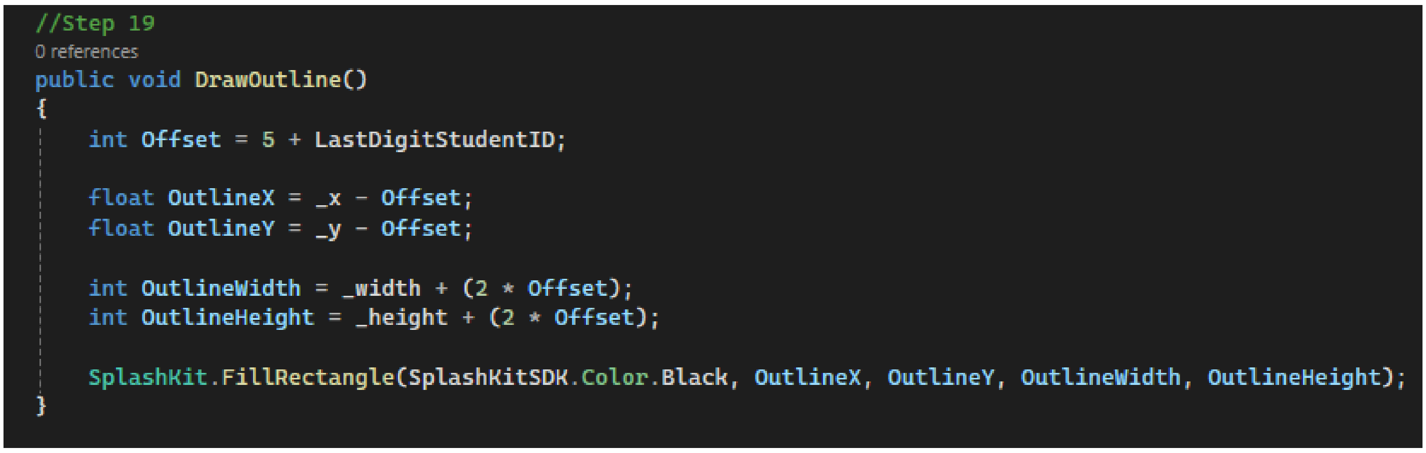


```
//Step 18
0 references
public List<Shape> SelectedShapes
{
    get
    {
        List<Shape> result = new List<Shape>();
        foreach (Shape s in _shapes)
        {
            if (s.Selected)
            {
                result.Add(s);
            }
        }
        return result;
    }
}
```

19. Switch back to class *Shape*. Add a ***DrawOutline*** method to the *Shape* class. This method draws a black rectangle around the shape. **The black rectangle has to be (5+X) pixels wider on all sides of the shape, where 'X' presents the last digit of your student ID.**

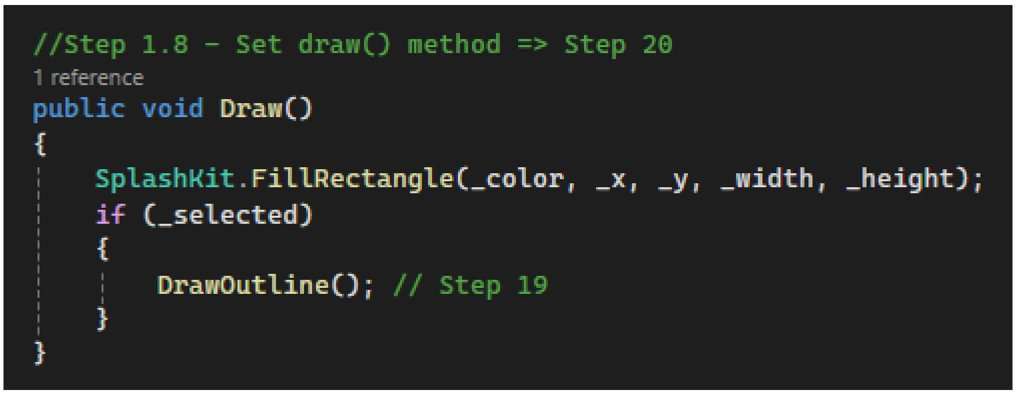


```
private const int LastDigitStudentID = 0; //Step 19
```



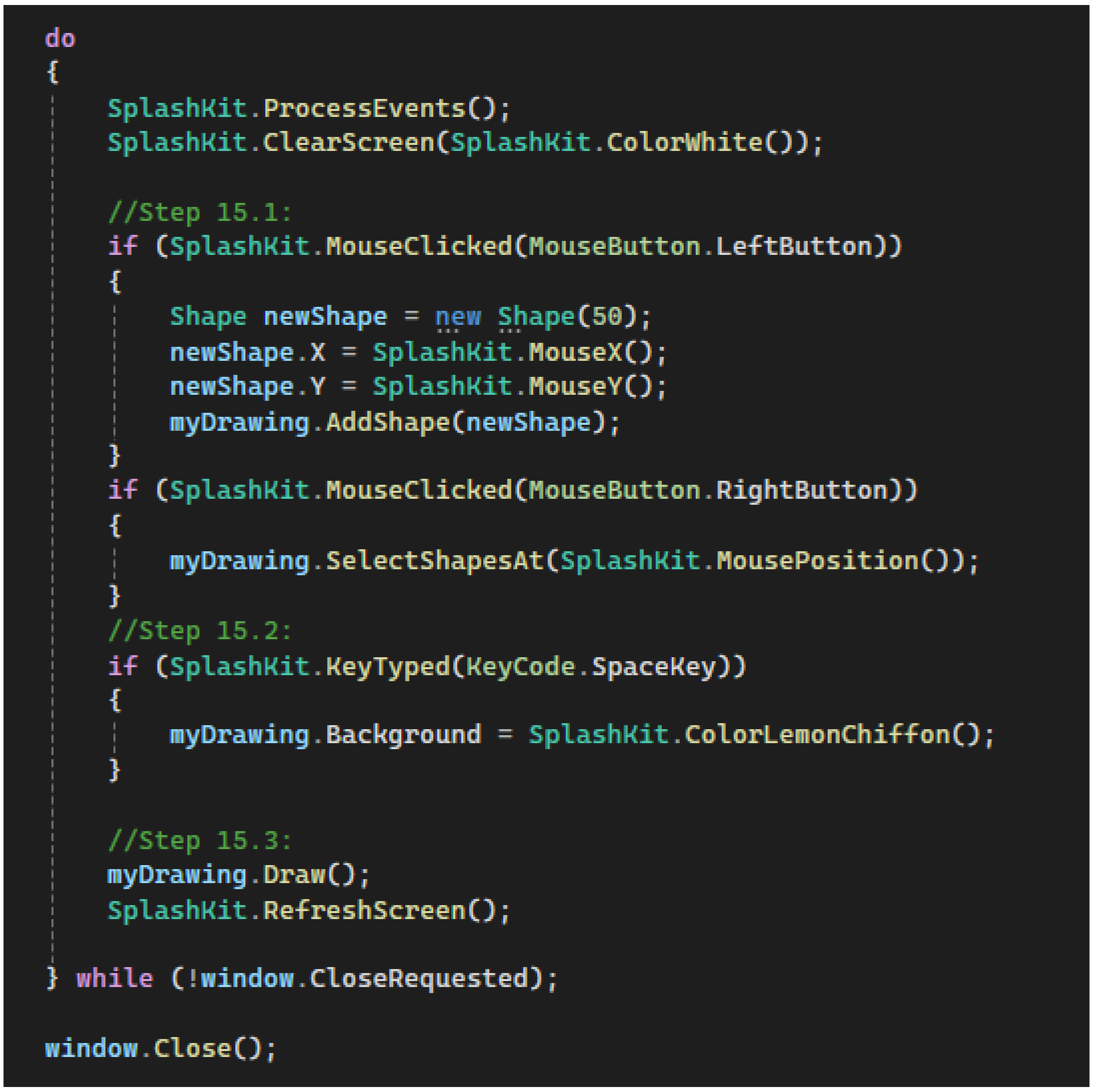
```
//Step 19
0 references
public void DrawOutline()
{
    int Offset = 5 + LastDigitStudentID;

    float OutlineX = _x - Offset;
    float OutlineY = _y - Offset;

    int OutlineWidth = _width + (2 * Offset);
    int OutlineHeight = _height + (2 * Offset);

    SplashKit.FillRectangle(SplashKitSDK.Color.Black, OutlineX, OutlineY, OutlineWidth, OutlineHeight);
}
```

20. Change the Shape's ***Draw*** method add some code to call ***DrawOutline*** if the shape is selected.



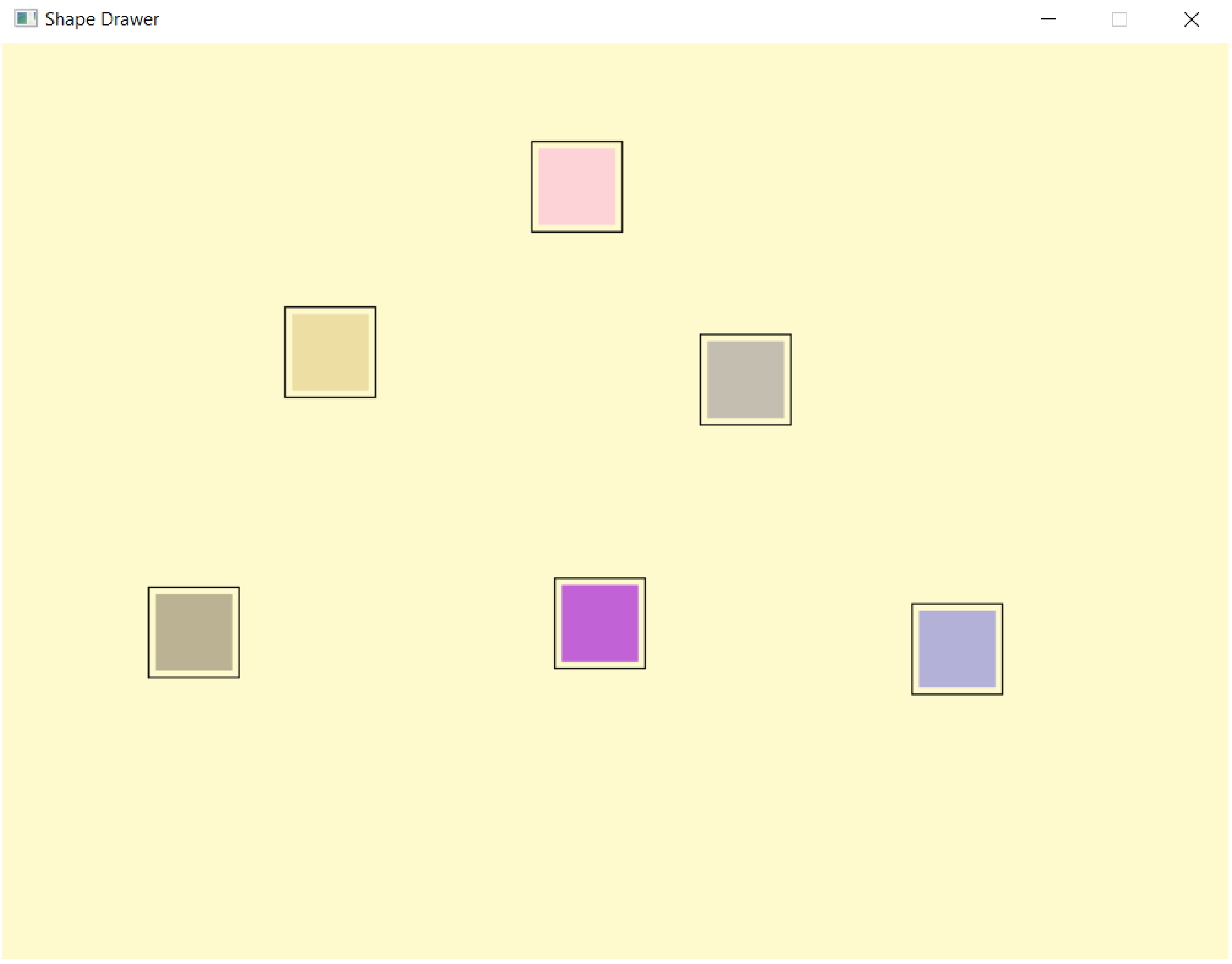
```
//Step 1.8 - Set draw() method => Step 20
1 reference
public void Draw()
{
    SplashKit.FillRectangle(_color, _x, _y, _width, _height);
    if (_selected)
    {
        DrawOutline(); // Step 19
    }
}
```

21. In function ***Main***, inside the event loop, add an ***if***-statement to check whether the user has clicked the right mouse button. If this is the case, tell ***myDrawing*** to **SelectShapesAt** the current mouse pointer position.

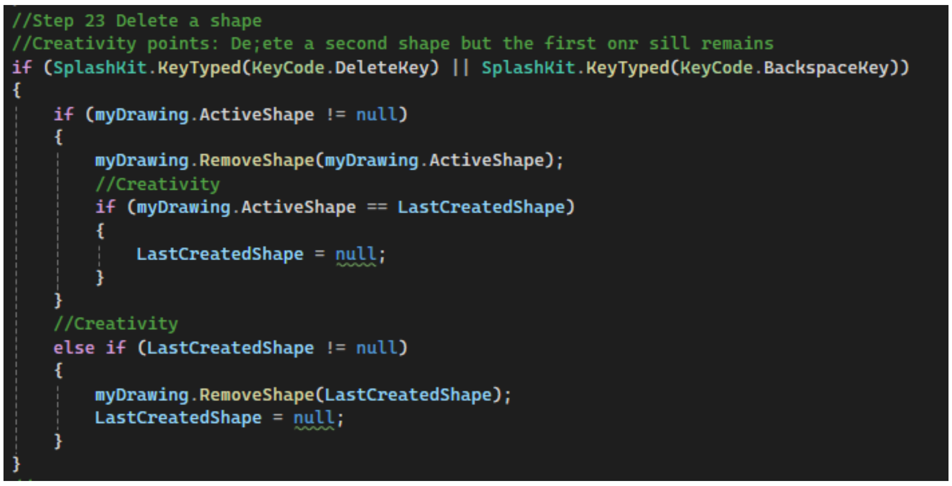


```
do
{
    SplashKit.ProcessEvents();
    SplashKit.ClearScreen(SplashKit.ColorWhite());

    //Step 15.1:
    if (SplashKit.MouseClicked(MouseButton.LeftButton))
    {
        Shape newShape = new Shape(50);
        newShape.X = SplashKit.MouseX();
        newShape.Y = SplashKit.MouseY();
        myDrawing.AddShape(newShape);
    }
    if (SplashKit.MouseClicked(MouseButton.RightButton))
    {
        myDrawing.SelectShapesAt(SplashKit.MousePosition());
    }
    //Step 15.2:
    if (SplashKit.KeyTyped(KeyCode.SpaceKey))
    {
        myDrawing.Background = SplashKit.ColorLemonChiffon();
    }

    //Step 15.3:
    myDrawing.Draw();
    SplashKit.RefreshScreen();

} while (!window.CloseRequested);

window.Close();
```

22. Compile and run the program and check that you can select shapes



23. Adjust the code so that all of the selected shapes are removed from the drawing if the user types the *KeyCode.DeleteKey* or *KeyCode.BackspaceKey*. Make sure your work is still consistent with the provided UML diagram — you do not need to add anything to class *Drawing* to make this work!



```
//Step 23 Delete a shape
//Creativity points: De;ete a second shape but the first onr sill remains
if (SplashKit.KeyTyped(KeyCode.DeleteKey) || SplashKit.KeyTyped(KeyCode.BackspaceKey))
{
    if (myDrawing.ActiveShape != null)
    {
        myDrawing.RemoveShape(myDrawing.ActiveShape);
        //Creativity
        if (myDrawing.ActiveShape == LastCreatedShape)
        {
            LastCreatedShape = null;
        }
    }
    //Creativity
    else if (LastCreatedShape != null)
    {
        myDrawing.RemoveShape(LastCreatedShape);
        LastCreatedShape = null;
    }
}
```

Note: Make sure to distribute the functionality across the program and class *Drawing*. The ***Main*** program should not interact with the class *Drawing*'s _***shapes*** list directly.

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: add shapes, select shapes, remove shapes, and change background.