|  |
| --- |
|  |
| C# Classes & Objects |
| Fun with Physics |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

# C# Classes and Objects

## Objectives

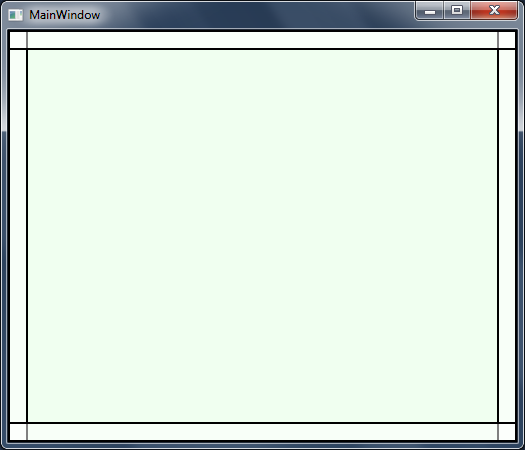
In this lab you’ll build classes and bring shapes to life in a physics simulation. By the time you complete the lab you should understand how to:

* Work with abstract base classes
* Override virtual methods
* Take advantage of polymorphism

This lab uses the [Farseer Physics Engine](http://farseerphysics.codeplex.com/) and the .NET Window Presentation Foundation library to build a desktop application.

## Part 1 – Laying the Foundation

1. Open the **Classy** project file from the **before** directory of this application.
2. Press **F5** to build and run the application. The application should run without error.



1. From the Solution Explorer window, open **Playground.cs**.

The Playground class represents the “canvas” or drawing area of the application. Notice this class derives from PhysicsCanvas and therefore inherits a number of capabilities. The PhysicsCanvas talks to a physics engine to simulate gravity and detect collisions.

Our goal is to add circles and squares to the drawing canvas. We’ll use a class to represent both types of shapes. Since the shapes will both need some common capabilities, we’ll use a base class to hold the common logic.

1. Right-click the project in Solution Explorer and select **Add -> New Class**. Name the file **ClassyShape**.

Normally we’d just name our classes Shape, Circle, and Rectangle even though both WPF and the physics engine have classes with these same names. The C# compiler won’t mind as long as all three “Circle” classes are in different namespaces. However, to avoid confusion in this lab we’ll name our classes ClassyShape, ClassyCircle, and ClassyRectangle.

1. Import the following **namespaces** at the top of the file.

using System.Windows.Media;  
using Classy.Physics;  
using FarseerGames.FarseerPhysics.Dynamics;  
using FarseerGames.FarseerPhysics.Mathematics;

1. Mark the ClassyShape class as **abstract** and **public.**

public abstract class ClassyShape  
{  
}

We don’t want the program to instantiate a shape – it’s only purpose is to serve as a base class. The program will create concrete classes derived from this base class.

Every shape needs to interact with the PhysicsCanvas at some point. We’ll force the program to give us a canvas using a constructor, and save a reference in a protected field so derived classes can access the canvas.

1. Add a **constructor** to the class taking a PhysicsCanvas parameter.
2. Save the incoming canvas into a protected field named \_canvas.

public abstract class ClassyShape  
{  
**public ClassyShape(PhysicsCanvas canvas)  
    {  
        \_canvas = canvas;  
    }  
  
    protected PhysicsCanvas \_canvas;**  
}

Every shape will also have a color.

1. Add a public property to ClassyShape named Color (and of type Color).

public abstract class ClassyShape  
{  
    public ClassyShape(PhysicsCanvas canvas)  
    {  
        \_canvas = canvas;  
    }  
  
**public Color Color { get; set; }**  
    protected PhysicsCanvas \_canvas;  
}

Properties in C# are a way to expose state from an object. In this case we allow external code to both read from (get) and write to (set) the property. In the next two steps we’ll set up common pieces all derived classes will need.

1. Add a second protected field to the class named \_body and of type Body.
2. Add an abstract method to the class named Show. The method returns void and takes no parameters.

The class should now look like the following.

public abstract class ClassyShape  
{  
    public ClassyShape(PhysicsCanvas canvas)  
    {  
        \_canvas = canvas;  
    }  
  
    public Color Color { get; set; }  
    public abstract void Show();  
  
    protected Body \_body;  
    protected PhysicsCanvas \_canvas;  
}

The Body type is defined by the physics engine and represents an object the physics engine manipulates. You can push a body around the screen by applying forces, and the engine can see if the body collides with other bodies. The Show method is abstract because it needs to create a visual on the screen. We don’t know how to create a visual for an abstract shape, but we’ll implement this method in a concrete type to create a circle in the next section.

## Part II – Drawing Circles

1. **Build** (F6) the project to make sure there are no errors before processing.
2. Right-click the project and select Add -> New Class. Name the class ClassyCircle.
3. Import the following namespaces at the top of the file.

using System.Windows.Media;  
using Classy.Physics;

1. Mark the class as public, and derive from ClassyShape.

public class ClassyCircle : ClassyShape  
{  
}

1. Build (F6) the project. The compiler should give you an error because ClassyCircle does not implement the Show method.
2. Override the Show method.

If you type “override” while inside ClassyCircle, Visual Studio will allow you to select a member to override using the arrow keys and the Enter button.

public class ClassyCircle : ClassyShape  
{  
    public override void Show()  
    {  
        throw new **NotImplementedException**();  
    }  
}

1. Try to build the project again. Now you should see an error about the ClassyShape constructor.

The error isn’t really in the ClassyShape constructor. The problem is how our ClassyCircle class derives from a ClassyShape, but the compiler doesn’t know how to create a ClassyShape when we instantiate a ClassyCircle. Because the base class requires a PhysicsCanvas object as a constructor parameter, we’ll have to give our circle a constructor, too, and pass along a canvas to construct the base clas.

1. Give ClassyCircle a constructor that takes a PhysicsCanvas parameter and forwards the parameter to the base class using the base keyword.

public class ClassyCircle : ClassyShape  
{  
    public ClassyCircle(PhysicsCanvas canvas) : base(canvas)  
    {  
    }  
  
    public override void Show()  
    {  
        throw new **NotImplementedException**();  
    }  
}

1. Build the project again. There should be no errors now.
2. Delete the line of code inside the Show method and replace it with the code shown below.

public override void Show()  
{  
    \_body = \_canvas.CreateCircleBody(24, (int)\_canvas.ActualWidth / 2, 40);  
    \_canvas.AddCircleToCanvas(\_body, Color, 24);  
}

This code is making use of the base class by referencing the \_canvas field and saving the newly created body into the \_body field. We’ll see how to use the \_body reference later in this lab.

1. Return to the Playground.cs file.
2. Add the following private field to the bottom of the Playground class.

private List<ClassyShape> \_shapes = new List<ClassyShape>();

We’ll use this list of shapes to keep track of all the shapes we create while the application is running.

1. Add a private method to the class named CreateCircle. The method returns void and takes no parameters.
2. Inside the method, create a new instance of the ClassyCircle class. Set the color of the class, add the object to the \_shapes list, and tell the object to Show itself.

You can pick any color you want by choosing one from the Colors class (like Colors.Red).

private void CreateCircle()  
{  
    ClassyCircle circle = new ClassyCircle(this);  
    circle.Color = Colors.Red;  
    \_shapes.Add(circle);  
    circle.Show();  
}

1. In the Playground class, override the OnKeyDown method.

Remember you can type “override” inside the class and let Visual Studio generate most of the following code for you.

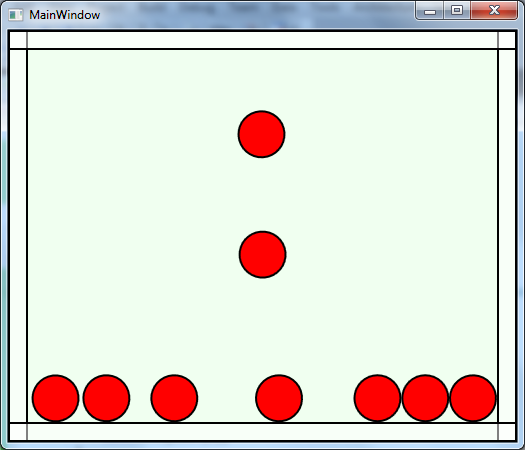
protected override void OnKeyDown(System.Windows.Input.KeyEventArgs e)  
{  
    base.OnKeyDown(e);  
}

1. Add the code shown below to the newly overridden method.

protected override void OnKeyDown(System.Windows.Input.KeyEventArgs e)  
{  
    switch (e.Key)  
    {  
        case Key.D2:  
            CreateCircle();  
            break;  
    }  
  
    base.OnKeyDown(e);  
}

When the program runs and someone presses the “2” key – a new circle should appear! Let’s try it.

1. Press F5 to build and run the project.
2. Once the application is open, press the 2 key a few times to make circles appear.



Once a circle is on the screen, you can click and drag it with the mouse to make it move around again.

## Part III – Rectangles

1. Add a new class to the project named ClassyRectangle.
2. Add the following using statements to the top of the file.

using System.Windows.Media;  
using Classy.Physics;  
using FarseerGames.FarseerPhysics.Mathematics;

1. Mark the class as public and derive from ClassyShape.

public class ClassyRectangle : ClassyShape  
{  
}

1. Just like we did with ClassyCircle, give the new class a constructor to take a PhysicsCanvas and provide an implementation of the Show method.

The class should look like the following:

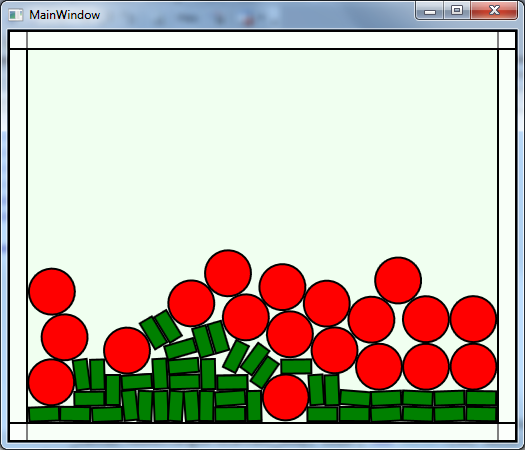
public class ClassyRectangle : ClassyShape   
{  
    public ClassyRectangle(PhysicsCanvas canvas)  
        : base(canvas)  
    {  
    }  
  
    public override void Show()  
    {  
        \_body = \_canvas.CreateRectangleBody(16, 32,

(int)\_canvas.ActualWidth / 2, 40);  
        \_canvas.AddRectangleToCanvas(\_body, Color, new Vector2(16, 32));  
    }  
}

1. Return to Playground.cs and add a private CreateRectangle method.
2. The CreateRectangle method should follow the same steps as CreateCircle (but use a ClassyRectangle).

private void CreateRectangle()  
{  
    ClassyRectangle rectangle = new ClassyRectangle(this);  
    rectangle.Color = Colors.Green;  
    \_shapes.Add(rectangle);  
    rectangle.Show();              
}

1. Inside of the Playground’s OnKeyDown method, add another case statement to assign the “1” key to create a rectangle.
2. protected override void OnKeyDown(System.Windows.Input.KeyEventArgs e)  
   {  
       switch (e.Key)  
       {  
   **case Key.D1:  
               CreateRectangle();  
               break;**  
           case Key.D2:  
               CreateCircle();  
               break;  
       }  
     
       base.OnKeyDown(e);  
   }
3. Press F5 to build and run the application.



## Part III – Polymorphism Makes You Jump!

In this section we’ll give all of our shapes the ability to jump back into the air.

1. Open ClassyShape.cs and add the following method to the ClassyShape class.

public void Jump()  
{  
    \_body.ApplyImpulse(new Vector2(250, -1000f));  
}

1. Open Playground.cs.
2. In the OnKeyMethod of Playground, add a third case statement to make the “3” key invoke a JumpShapes method.

                case Key.D3:  
                    JumpShapes();  
                    break;

1. Create a private method in Playground named JumpShapes (Visual Studio will do this for you if you put the cursor over JumpShapes in the switch statement, press Ctrl + ., and select “Generate …”.
2. Inside JumpShapes, loop through the \_shapes collection and invoke Jump on every shape.

private void JumpShapes()  
{  
    foreach(ClassyShape shape in \_shapes)  
    {  
        shape.Jump();  
    }  
}

1. Press F5 to build, run, and test the Jump functionality.

Notice how all the shapes jump up and slightly to the right.

1. Open ClassyRectangle.cs
2. Add a Jump method to the class with the following code.

public void Jump()  
{  
    \_body.ApplyImpulse(new Vector2(-250, -1000f));  
}

1. Press F5 to run the application.

Make different objects jump around. Notice they still all most to the right. Do you know why? See if you can make rectangles jump slightly to the left without using the following steps.

1. Open ClassyShape.cs and add the virtual keyword to the Jump method.

public virtual void Jump()  
{  
    \_body.ApplyImpulse(new Vector2(250, -1000f));  
}

1. Open ClassyRectangle and add the override keyword to the Jump method.

 public override void Jump()  
 {  
     \_body.ApplyImpulse(new Vector2(-250, -1000f));  
 }

1. Press F5 to test the application again…

## Conclusion

Congratulations! You’ve used the foundational object oriented programming techniques in the C# language. For extra credit you can continue to add new classes or functionality to the application. Create shapes with different sizes, different colors, and different behaviors.