When and How to Make Something a Class

Before we get started, in this lesson we're going to use the project we created in the "Faster Graphics" lesson. So if you haven't done it, or you discarded the project (shame on you), you need to go back and do "Faster Graphics" lesson first. ☺

Okay, what we're going to learn today is determining when it makes sense to put something in its own class.

Open your *UserPaintExample* project from the *Faster Graphics* lesson. As a review, what we did was:

1. Take user control of painting the window
2. Created a ball that we moved across the screen and bounced off the edges

Now if you stop and think about it, there are two basic groups of "stuff" in that program:

1. The ball specific stuff
2. and the *Form1* (GUI) specific stuff

The way things are currently setup, all of the ball stuff is buried in and sprinkled throughout the *Form1* class. All the properties of the ball (its size, color, location, movement info, etc) are fields in the *Form1* class. The code for moving the ball and drawing it is directly part of *Form1* class methods. Put another way, the ball is an integral part of the form.

C# is an object oriented language. What does that mean? It means that things in C# are objects. There are different types of objects: there are forms, there are buttons, there are timers … and in this case there is also this thing called a "ball."

This object stuff may not sound like a very big deal, but it actually is *very* powerful. Why? Because generally we don't need to know how something works in order to use it. Consider your cell phone: I'd bet most people could care less how it works, they simply want to know how to use it.

You used many objects in the *UserPaintExample* project: *Timer*, *Point*, *Size, Pen, Brush* to name a few. Do you know how the *Timer* works? How about the *Brush*? Nope you really don't. But that doesn't matter because you know how to use them.

The power of objects is that we can package something complex up in an object (class) and as long as it has the features we need, we can simply use it. And more to the point, we can reuse it at will.

Not convinced? Well, try this on for size…

When and How to Make Something a Class

Suppose I gave you the task of adding 20 balls to your *UserPaintExample* project so you'd have a bunch of balls moving on the screen and bouncing off the sides. How would you do this?

Well, one obvious way to tackle this would be to copy all the ball stuff 20 times. You've have 20 different location *Points*, 20 different sizes, 20 different dX's and 20 different dY's. You've have to duplicate the move and bounce code that is in the *refreshTimer_Tick()* method 20 different times.

What a nightmare, what a mess … makes my head hurt just thinking about it.

This just ***SCREAMS*** for ball becoming an object … for making a class for it. Once we have a class for it, we can make as many instances of it as we want and hold each instance in a variable. That one variable will contain the entire object ball. It will hold its own copy of the size, location, dX, dY, etc.

If ball were its own class (let's call that class ***Ball***) we could make a field to hold the ball like this:

```
private Ball ball;
```

Then once we had this field (which is of type `Ball`, and whose name is "ball") we could then use whatever methods are available (public) in the `Ball` class. Suppose there was a Move() method in this `Ball` class. Then we could simply say:

```
ball.Move();
```

So what you say? Well, remember that the process of moving the ball in the *refreshTimer_Tick()* method was something like 22 lines of code. This now lets us move a bunch of balls very easily:

```
ball1.Move();
ball2.Move();
ball3.Move();
ball4.Move();
ball5.Move();
ball6.Move();
```

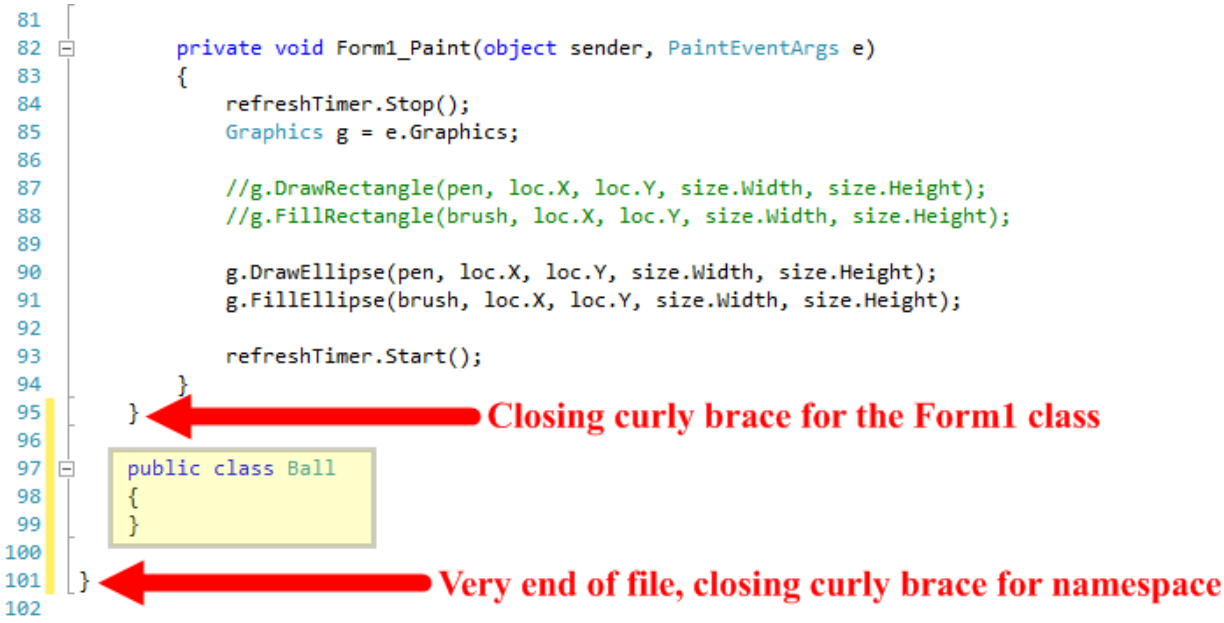
Or even better … suppose we'd put all the balls into an array called balls. We could then use a loop to move all the balls:

```
for (int i = 0; i < balls.Length; i++)
{
    balls[i].Move();
}
```

Why is that cool? What if we wanted to have 100 balls on the screen? Just put all 100 balls into the array and the above loop will move them all! And it only took 4 lines of code. Cool!

When and How to Make Something a Class

Okay, so we should make a class for ball. Let's do it! First step, create the Ball class. We can add the code right after the *Form1* class. Don't create the *Ball* class inside the *Form1* class. If you do we won't be able to use the *Ball* class anywhere else. Make it outside of the *Form1* class.



Here you can see I added the new *Ball* class down at the bottom of the file, ***after*** the closing curly brace for the *Form1* class. It is an empty class, nothing in it.

When and How to Make Something a Class

What we need to do now is move all the ball specific stuff from the *Form1* class down into our new *Ball* class. Start with all the ball specific fields: loc, size, pen, brush, dX and dY. Make sure you ***move*** them; don't copy them and leave copies behind in the *Form1* class. After doing this here is what the *Ball* class looks like:

```
86      public class Ball
87      {
88          private Point loc;
89          private Size size;
90
91          private Pen pen;
92          private Brush brush;
93
94          private int dX;
95          private int dY;
96      }
```

…and here is what the top of the *Form1* class now looks like … it only has a *Timer* field right now:

```
10  namespace UserPaintExample
11  {
12      public partial class Form1 : Form
13      {
14          private Timer refreshTimer;
15
16          public Form1()
17          {
18              InitializeComponent();
```

All the ball specific fields used to be here.

When and How to Make Something a Class

Next we need to move the initialization of the ball fields from the *Form1* constructor to the *Ball* constructor. To do this we need to create a constructor for the *Ball* class. Remember, the constructor is the special method that gets called to create an instance of the class. It has no return type and its name is the same as the class. Here is the empty constructor…add this code:

```
86      public class Ball
87      {
88          private Point loc;
89          private Size size;
90
91          private Pen pen;
92          private Brush brush;
93
94          private int dX;
95          private int dY;
96
97          public Ball()
98          {
99          }
100     }
```

When and How to Make Something a Class

Now move the lines of code in the *Form1* constructor that initializes all the ball fields into the new *Ball* constructor. Here is what the new constructor looks like:

```
78      public class Ball
79      {
80          private Point loc;
81          private Size size;
82
83          private Pen pen;
84          private Brush brush;
85
86          private int dX;
87          private int dY;
88
89          public Ball()
90          {
91              // initialize the critical ball information
92              loc = new Point(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
93              size = new Size(15, 15);
94              brush = new SolidBrush(Color.Blue);
95              pen = new Pen(brush);
96              dX = 4;
97              dY = 6;
98          }
```

Hmm, we have the dreaded red squiggly lines under the word *ClientRectangle* in the moved code. Why is that? Well, because *ClientRectangle* is a thing that is owned by the GUI, by *Form1*. and *Ball* doesn't know what it is. In order to determine how to fix this, we need to answer two questions.

1. First, what is *ClientRectangle*? It is the object that represents the screen information including where the edges are, how big it is, etc.
2. Second, what does *Ball* need it for? Well here (in the constructor) *ClientRectangle* is being used to place the ball in the center of the window area.

In this situation, really all we need to know is where to place the ball. We can handle that by passing into the constructor two parameters: the X and the Y coordinates at which to place the ball! Remember, a parameter is nothing more than a variable that is used to receive incoming information for a method, and it is placed/created inside the parenthesis of the method name:

```
91          public Ball(int x, int y)
92          {
93              // initialize the critical ball information
94              loc = new Point(x, y);
95              size = new Size(15, 15);
96              brush = new SolidBrush(Color.Blue);
97              pen = new Pen(brush);
98              dX = 4;
99              dY = 6;
100         }
```

This is actually very helpful because it gives us great flexibility over where a ball appears when it is created.

When and How to Make Something a Class

Next, we need to get the ball movement and bounce code out of the *refreshTimer_Tick()* method in *Form1*. Let's make a Move() method in the *Ball* class and move all that code into it. I added the new Move() method after the *Ball* constructor method:

```
78          public void Move()
79          {
80              // move the ball
81              loc.X += dX;
82              loc.Y += dY;
83
84              // keep the ball on the screen, and bounce off
85              if (loc.X < 0)
86              {
87                  loc.X = 0;  // hit left edge
88                  dX = -dX;
89              }
90              else if (loc.X + size.Width > ClientRectangle.Width)
91              {
92                  loc.X = ClientRectangle.Width - size.Width;  // hit right edge
93                  dX = -dX;
94              }
95              else if (loc.Y < 0)
96              {
97                  loc.Y = 0;  // hit top edge
98                  dY = -dY;
99              }
100             else if (loc.Y + size.Height > ClientRectangle.Height)
101             {
102                 loc.Y = ClientRectangle.Height - size.Height;  // hit bottom edge
103                 dY = -dY;
104             }
105         }
106     }
```

And here is what the *refreshTimer_Tick()* method looks like in *Form1*:

```
31          private void refreshTimer_Tick(object sender, EventArgs e)
32          {
33              // this is where the ball move/bounce code was...
34
35              // tell windows to repaint the window
36              Invalidate();
37          }
```

Please note the comment on line 33 … this is where the ball move and bounce code was. We want to remember this.

When and How to Make Something a Class

Now if you look back at the new *Move()* method in the *Ball* class, you'll that the red squiggly lines are showing up again under the references to *ClientRectangle*. Same problem as before, very similar solution. Again, the same question: what does the *Ball Move()* method need *ClientRectangle* for?

In this case it needs it to determine where the edges of the "box" are so it can bounce. We will provide this information via a parameter to the *Move()* method. In this case we'll simply pass in the whole *ClientRectangle*. To do that we need to figure out what type of thing *ClientRectangle* is. It is a *Rectangle* object; makes sense doesn't it? ☺ Here are the code changes:

```
78          public void Move(Rectangle boundingBox)
79          {
80              // move the ball
81              loc.X += dX;
82              loc.Y += dY;
83
84              // keep the ball on the screen, and bounce off
85              if (loc.X < 0)
86              {
87                  loc.X = 0;  // hit left edge
88                  dX = -dX;
89              }
90              else if (loc.X + size.Width > boundingBox.Width)
91              {
92                  loc.X = boundingBox.Width - size.Width;  // hit right edge
93                  dX = -dX;
94              }
95              else if (loc.Y < 0)
96              {
97                  loc.Y = 0;  // hit top edge
98                  dY = -dY;
99              }
100             else if (loc.Y + size.Height > boundingBox.Height)
101             {
102                 loc.Y = boundingBox.Height - size.Height;  // hit bottom edge
103                 dY = -dY;
104             }
105         }
```

So now to use the *Move()* method in *Ball* you need to tell it what rectangle (the more technical term for this is bounding box) it is moving in.

When and How to Make Something a Class

We now have one bit of code left … that is the two lines of code in *Form1_Paint* that draws the ball. We need to move this. Let's create a *Draw()* method in the *Ball* class for this and move the two lines of code:

```
102         public void Draw()
103         {
104             g.DrawEllipse(pen, loc.X, loc.Y, size.Width, size.Height);
105             g.FillEllipse(brush, loc.X, loc.Y, size.Width, size.Height);
106         }
107     }
108
109 }
```

Closing curly brace for the Ball class

It's very difficult to see but we have red squiggly lines under the ***g*** on lines 104 and 105. This is because the *Ball* class doesn't know what ***g*** is. If you look back at the *Form1_Paint()* method this came from, ***g*** is the graphics context in which to draw. We can easily solve this problem by passing the graphics context in as a parameter. What type is ***g***? It is a *Graphics* object. So here ya go:

```
102         public void Draw(Graphics g)
103         {
104             g.DrawEllipse(pen, loc.X, loc.Y, size.Width, size.Height);
105             g.FillEllipse(brush, loc.X, loc.Y, size.Width, size.Height);
106         }
```

Super simple! And we now have a complete *Ball* class ready to use!

All we need to do now is go back to the *Form1* class and use our new *Ball* class.

When and How to Make Something a Class

First we'll add a field to *Form1* to hold our ball:

```
12      public partial class Form1 : Form
13      {
14          private Timer refreshTimer;
15
16          private Ball ball;
17
18          public Form1()
19          {
20              InitializeComponent();
```

Next we'll initialize it (create it) in the *Form1* constructor:

```
12      public partial class Form1 : Form
13      {
14          private Timer refreshTimer;
15
16          private Ball ball;
17
18          public Form1()
19          {
20              InitializeComponent();
21
22              // Tell windows we are taking responsibility for painting the screen
23              SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.UserPaint |
24              this.Paint += new PaintEventHandler(Form1_Paint);
25
26              // Create the ball and place it in the middle of the screen
27              ball = new Ball(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
28
29              // Screen refresh timer
30              refreshTimer = new Timer();
31              refreshTimer.Interval = 25;
32              refreshTimer.Tick += new EventHandler(refreshTimer_Tick);
33              refreshTimer.Enabled = true;
34          }
```

When and How to Make Something a Class

Now in the *refreshTimer_Tick()* method where we used to have the ball move/bounce code, we'll add a call to the ball *Move()* method. Remember, we need to pass in the *ClientRectangle* to the *Move()* method:

```
36          private void refreshTimer_Tick(object sender, EventArgs e)
37          {
38              // this is where the ball move/bounce code was...
39              ball.Move(ClientRectangle);
40
41              // tell windows to repaint the window
42              Invalidate();
43          }
```

The final step is to add a call to the ball *Draw()* method in *Form1_Paint()* where we used to draw the ball. Here we need to pass in the graphics context:

```
45          private void Form1_Paint(object sender, PaintEventArgs e)
46          {
47              refreshTimer.Stop();
48
49              ball.Draw(e.Graphics);
50
51              refreshTimer.Start();
52          }
```

When and How to Make Something a Class

We're done! If you build and run your program you should see the ball moving as before. What did all this buy us? Well for one, look at your *Form1* code … it is much simpler and more compact. Much easier to read and understand what is happening.

For another, it is now trivial to add another ball to the program…all you have to do is:

1. add another *Ball* field
2. create it in the *Form1* constructor deciding where you want it to appear
3. add a call to the new ball's *Move()* method in *refreshTimer_Tick()*
4. and add a call the new ball's *Draw()* method in *Form1_Paint()*

In total that amounts to 4 lines of code! As a quick reference, here is some of the code, placing the new ball at (25, 100) for its starting position:

```
27              // Create the ball and place it in the middle of the screen
28              ball = new Ball(ClientRectangle.Width / 2, ClientRectangle.Height / 2);
29              ball2 = new Ball(25, 100);
30
31              // Screen refresh timer
32              refreshTimer = new Timer();
33              refreshTimer.Interval = 25;
34              refreshTimer.Tick += new EventHandler(refreshTimer_Tick);
35              refreshTimer.Enabled = true;
36          }
37
38          private void refreshTimer_Tick(object sender, EventArgs e)
39          {
40              // this is where the ball move/bounce code was...
41              ball.Move(ClientRectangle);
42              ball2.Move(ClientRectangle);
43
44              // tell windows to repaint the window
45              Invalidate();
46          }
```

Also add the field and the call to *ball.Draw()* in *Form1_Paint()*, run and you should now have 2 balls moving independently on the screen, and you only had to add 4 lines of code. Ah the wonders of objects and classes! ☺