Inft2012 Application Programming

Lecture 10

Classes and objects

Creating our own class

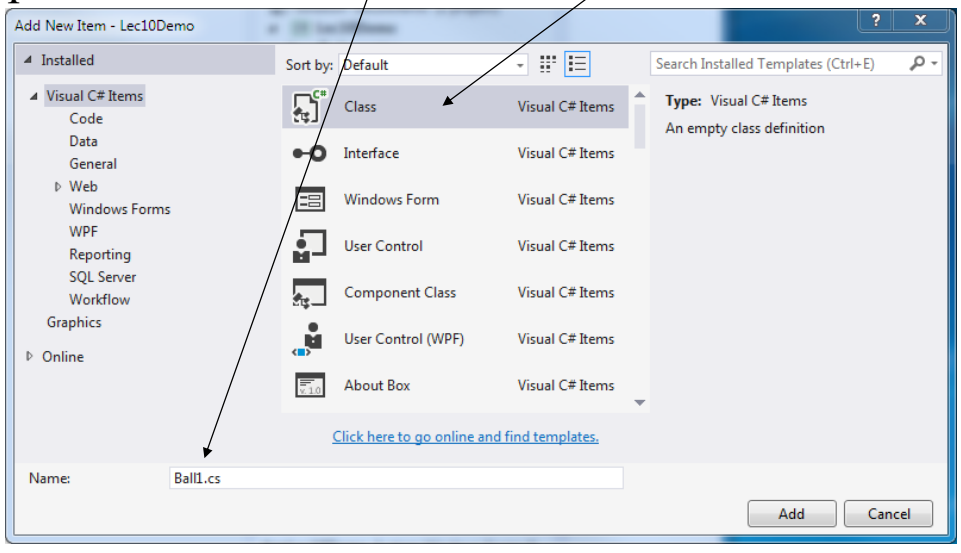
- We have declared and used objects of many pre-defined classes, such as
  - Form, Button, TextBox
  - Graphics, Pen, SolidBrush
  - Math.Random
- (Though some of the declaration has been done for us)
- We've been telling you how to *use* objects and classes since the start, but we've left it till now to explain how you can *create* classes
- This material ties in with a concept called object-oriented programming

What is a class?

- A class is the definition of a set of *potential* objects
- Objects have attributes, so a class will define attributes
- Objects have methods, so a class will define methods
- We define a class by writing code to implement its methods and attributes
- Once we've written the class, we can declare objects, instances, of that class . . .
- . . . just as we currently declare objects of classes such as Form, Button, TextBox, etc
- We'll start with a nice simple class – a graphical ball

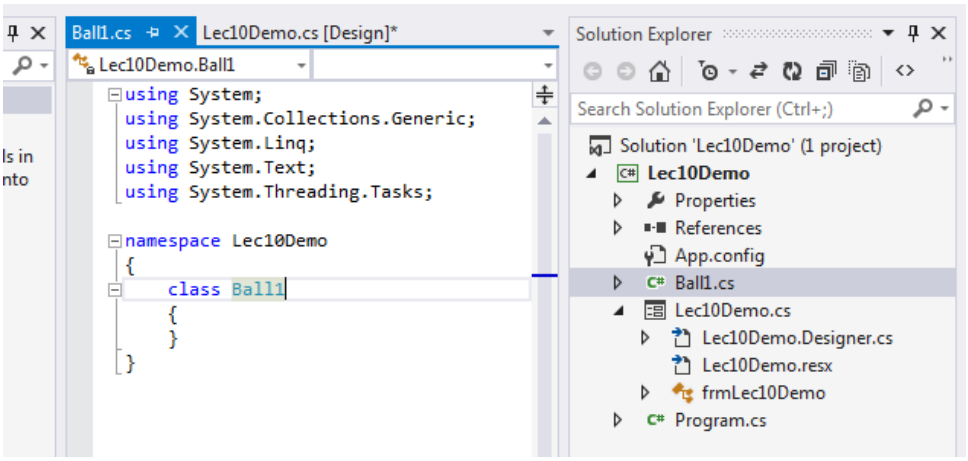
Creating a new class

- To create the sort of class we're talking about here, select Project / Add Class, be sure that Class is selected, and enter a sensible name for the class – starting with a capital letter



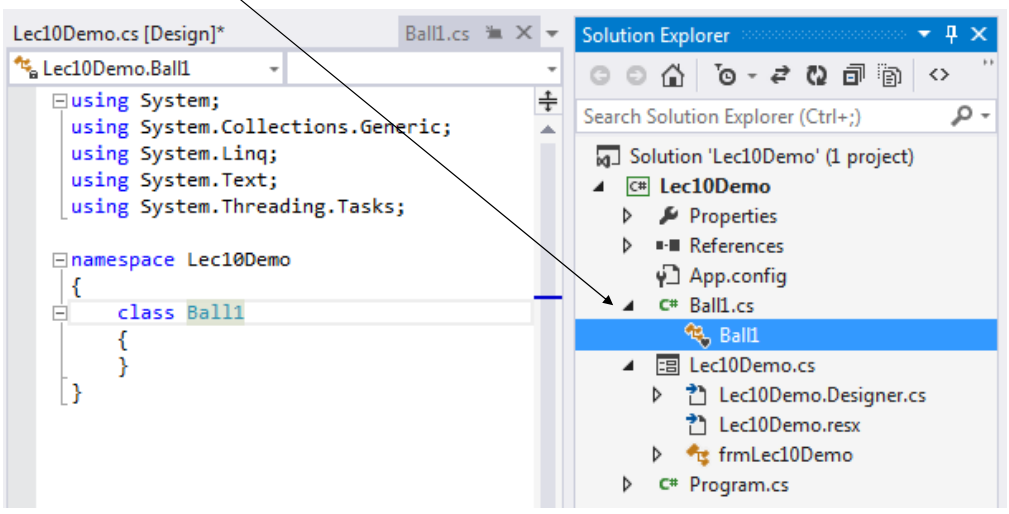
A new class

- Your new class shows up in Solution Explorer, and you are shown some empty code
- The class is in a separate file from the code for your form, so start with the usual comments, saying who wrote it, and when, and why



Displaying the code

- If you later open the project and can't see the code, expand the class, then click the code icon below it



Attributes or instance variables

- A ball (in two dimensions, a circle) has a colour, the location of its centre, and a radius
- We'll start by making them public (within the braces of class Ball)

```
public int iAcross, iDown, iRadius;
public SolidBrush sbColour;
```

- Our ball is one class, and the form (which will use the ball) is a different class; making the ball attributes public means that the form can access them
- Note that C# doesn't recognise SolidBrush, but is still very helpful: clicking on it gives a couple of options, one of which is to add *using System.Drawing;* if you select this, the line is added in the right place

Method

- We also require one method, which draws the circle
- This also goes within the class braces, after the instance variables

```
public void Draw(Graphics graPaper)
{
  graPaper.Clear(Color.White);
  graPaper.FillEllipse(sbColour, iAcross – iRadius,
    iDown - iRadius, iRadius * 2, iRadius * 2);
}
```

- Remembering the way C# draws circles, the Ball1 class converts the centre and radius of the circle to the top left corner and size of its enclosing square

Method parameter

- The drawing methods such as FillEllipse are methods of a Graphics object
- When defining the Ball1 class, we don't know which Graphics object will be used, so we specify it as a parameter
- When our form Ball1Demo.cs calls the draw method of our Ball class, it will have to provide the appropriate Graphics object as an argument

Our form and program

- Examine the code for the Ball 1 demo form in Lec10Demo
- Note where it declares the Ball1 object balRed
- The form's constructor (public Ball1Demo) associates a graphics object with the picture box,
- then instantiates balRed and gives initial values to its attributes
- The form has a button to display the ball, buttons to move it up, down, left, and right, and buttons to make it grow or shrink
- All of these buttons call the ball's Draw method; all but the first alter the appropriate attribute first

Exploring the program

- Explore the program in action
- If you want a button to repeat, click the button once to select it, then hold down the Enter key
- Do we really want the ball to be able to disappear off the edges of the picture box?
- Do we really want its diameter to be able to go negative or huge?
- What choice do we have? Surely that's up to the user?

Private attributes, public properties

- If attributes (instance variables) are public, other classes (such as our form, and therefore its users) can give them whatever values they like
- Now we're going to make them private, and associate them with public properties
- A property looks from the outside just like an attribute
- But when we assign to or from a property, it isn't a direct assignment . . .
- . . . instead it's 'controlled' by going through programmer-defined methods called *get* and *set*

Assignments use get and set

- The *get* method (a function, which returns a value) is used to provide (read) the value of the property
- When we type

```
<variable> = <object>.<property>;
```

  C# does

```
<variable> = <object>.<property>.get();
```

- The *set* method is used to alter (write) the value of the property
- When we type

```
<object>.<property> = <value>;
```

  C# does

```
<object>.<property>.set(<value>);
```

- (Well, not exactly, but we'll soon see the difference)

Naming attributes and properties

- Properties are seen from outside the class; they are what the rest of the program sees and works with
- Private attributes are seen only within the class, and can be accessed only by that code
- A property and a corresponding attribute generally represent the same quantity, but they need different names so as not to be confused
- One possible convention, which we'll use, is to give the attribute the same name as the property, but beginning with an underscore
- The underscore tells the programmer: 'this is the private version'

Creating properties

- To create a property we type
  `public <type> <name>`
  for example, `public int iAcross`
- But we don't add a semi-colon as we would when declaring a variable; a property definition includes one or two method definitions, which are enclosed in braces

```
// Private attributes of a ball
private int _iAcross, _iDown, _iRadius;
private SolidBrush _sbColour;

// Public properties to access those attributes
10 references
public int iAcross  // property corresponds to attribute _iAcross
{
    get
    {
        return _iAcross;
    }
    set
    {
        _iAcross = value;
    }
}
```

This is the simplest code for get

This is the simplest code for set

The magical 'value'

- Remember, assignments use *set* and *get*
- When the program says `balRed.iAcross = 15;` , C# does `balRed.iAcross.set(15);` that is, it executes the *set* method with an argument of 15
- Every argument should have a corresponding parameter, but *set* doesn't seem to have one
- In fact it has a hidden parameter called *value*, of the same type as the property
- So within the *set* method, the name *value* always refers to the argument, the value that's being assigned to the property

Why bother with properties?

- The point of properties is that the get and set methods can be used to exercise control that isn't available with simple assignment to public attributes
- For example, we can use properties to ensure that our ball doesn't go outside the picture box and doesn't get less than 10 pixels in diameter
- The simplest code shown two slides back offers no control at all; each method just passes on the value

A property with control

- Let's see what a property can do that an attribute can't
- We've kept the meaningful name *iAcross* for the public property, changing the now-private attribute to *_iAcross*

```
public int iAcross  // property corresponds to attribute _iAcross
{
    get
    {
        return _iAcross;
    }
    set
    {
        // Ensure that the ball can't leave the paper
        if (value < iRadius) _iAcross = iRadius;
        else if (value > iPaperSize - iRadius) _iAcross = iPaperSize - iRadius;
        else _iAcross = value;
    }
}
```

Properties in the Ball class

- The Ball2 class has private attributes and public properties
- The Ball 2 demo form illustrates the use of properties with the Ball2 class
- Because we've used the same names for the properties that we earlier used for the attributes, the code in Ball2Demo is exactly the same as in Ball1Demo
- But because we've used the *set* methods to control the range of values, the program's behaviour is quite different
- Examine the code and examine the program's behaviour

Read-/write-only properties

- We can exert even more control through properties
- If we give a property no *set* method, it will be read-only; it can't be changed from outside the class
- If we give a property no *get* method, it will be write-only; it can be changed but not seen from outside the class

```
public string sName
{   // Attribute _sName, read-only
    get
    {
        return _sName;
    }
}
```

```
public string sPassword
{   // Attribute _sPassword, write-only
    set
    {
        _sPassword = value;
    }
}
```

Data hiding

- We can't access the attributes of a Ball object from our form (which is outside Ball) because we made them private
- We did this to implement *data hiding*, the principle that attributes of a class should only be accessible from within that class
- We should always have private attributes accessed via public properties, even if all the properties do is echo the attributes
- This is a feature of good programming

Constructors

- When we instantiate a declared object using *new*

```
Ball balRed = new Ball();
```

or

```
SolidBrush sbColour = new
  SolidBrush(Color.Red);
```

we are using a special method called a *constructor*

- One constructor, with no parameters, is automatically provided for any class; all it does is create (*declare* and *instantiate*) a new instance (individual object) of that class
- You can write your own constructor(s), with whatever parameters you think are appropriate; these will typically be used to *declare*, *instantiate*, and *initialise* a new instance of the class

If Ball had a better constructor

- Our form includes the code

```
balRed = new Ball2();
balRed.iAcross = 100;
balRed.iDown = 200;
balRed.iRadius = 50;
```

- This might look better as

```
balRed = new Ball2(100,200,50);
```

- We can arrange this by writing a constructor that takes 3 integer arguments and uses them to set the _iAcross, _iDown, and _iRadius attributes
- Better still, as the *properties* make sure the *attributes* have sensible values, we'll have our constructor set the iAcross, iDown, and iRadius *properties*

A constructor for Ball

- Our constructor would be written . . .

```
public Ball3(int x, int y, int iRad)
{
   // Constructor with location and radius;
   // set a default colour
   sbColour = new SolidBrush(Color.Red);
   iAcross = x;
   iDown = y;
   iRadius = iRad;
}
```

- How do we (and Visual Studio) know it's a constructor? Because it has the same name as the class, and no type – not even void.

Multiple constructors

- We might want another constructor that sets the colour of the ball as well as its location and size . . .

```
balBall = new Ball3(Color.Red, 50, 90, 80);
```

- We can write as many constructors as we like, so long as they all have different *signatures*, different numbers of parameters and/or different types of parameters
- That is, we can *overload* the constructors
- Once we've written any constructor, the default constructor with no parameters is no longer available; if we still want that one as well as the one we've written, we have to write it too
- If we just want it to create the object, not to create and initialise it, no code is needed in the constructor body

Private methods

- Sometimes we might want to write methods that are to be used within the class, but not by anything outside the class
- For example, every time we draw the ball, we want to draw a black outline round it
- This doesn't really need a separate method, but if it were more complicated it might . . .
  . . . and the method would be called only by the class's own Draw method, not from the form . . .
  . . . so we'd make it private, and the form wouldn't even know it exists
- Ball3Demo illustrates multiple constructors and two private methods

Multiple objects

- If we decided to have several balls, but only one on the paper at a time . . .
- . . . we could declare several balls . . .
- . . . use buttons to select the one we want . . .
- . . . and have the controls operate whichever one was selected
- Take a close look at Ball4Demo
- Just for fun, we've added a timer, and can set the ball bouncing
- Whenever you use a timer, be sure to work out an appropriate interval for it; the default isn't always ideal

Things to notice

- Apart from the continuous movement, note that more of the detailed ball control code has been put in the Ball4 class, leaving less detail in the Ball4Demo (form) class
- If moving left, say, is perceived as something a ball does, it should be included as a method of Ball; the form class shouldn't have to work out how to do it
- Question to ponder: the form doesn't use the properties any more, it just uses the public constructors and draw methods; should even the properties be made private?
- If there's really no requirement to access them from outside, you could do this; but the convention is for them to be public

An improvement on parallel arrays

- Remember parallel arrays?
- For example, people's names in one array, their heights in a second array, their weights in a third array . . .
- We promised a better way of doing this, and here it is!
- We define a Person class, with a name, a height, a weight . . .
- . . . then create an *array* of Person objects
- There's no longer an issue of ensuring that the arrays remain parallel (eg when sorting the name array alphabetically, remembering to perform the same shuffle on the other arrays) because each person is self-contained

Array of objects – example

- The Array of person objects form illustrates this Person class and an array of Person objects
- It doesn't do much – it doesn't permit the array members to be sorted or shuffled, it doesn't permit heights and weights to be changed . . .
- but it should be enough to give an idea of how these features might be added if required
- We'll see a more thorough example next week

Object arrays and instantiation

- A single person might be declared and instantiated

```
Person perStudent = new Person();
```

- With an array of objects we can't combine these two operations; we must first declare and instantiate the array . . .

```
Person[] perGroup = new Person[26];
```

- . . . but we must then instantiate each individual person object inside some method

```
for (int i = 1; i < 26; i++)
    {  perGroup[i] = new Person();  }
```

- Alternatively, we can instantiate each one as it is needed – but *don't forget*, or the program will crash