Inft2012 Application Programming

Lecture 11

More on classes and objects

Revision: arrays

- An array is how we store many variables under one variable name, using an index to distinguish between them
- The power of the array lies in the index, because we can use a variable in a loop as the index, and thus process hundreds or thousands of array elements with just a few lines of code
- An array with n elements has indexes ranging from 0 to n–1
- If you find this confusing, consider giving the array n+1 elements and ignoring the element with index 0

Revision: classes

- A class is the definition on which an object is based; it's a template for objects
- Classes include attributes (generally private, and accessed through public properties) and behaviour (defined through public methods)
- C# already has classes defining many of the objects we want to use (forms, buttons, textboxes, etc)
- When we want to include other sorts of object in our programs (balls, students, etc), we write our own classes to define them
- The classes we write will include attributes, properties, and methods

Revision: objects

- When we use an object in a program we must . . .
- declare it – tell C# we want to use this name to refer to an object of this class
```
Ball balYellow;
```
- instantiate it – tell C# which particular object of this class we want the name to refer to
```
balYellow = new Ball();
```
- initialise it – tell C# what values to assign to the attributes of this object
```
balYellow.colColour = Color.Yellow;
balYellow.iSize = 48;
```
- These steps can be combined in various ways

Arrays in objects

- So far, we have used only numbers, strings, and booleans as attributes of our objects
- Attributes can also include arrays
- For example, a class to represent children in a health study might include the child's height at each year of age

```
chldCurrent.dHeightAtAge[0] = 53;
chldCurrent.dHeightAtAge[1] = 79;
chldCurrent.dHeightAtAge[2] = 95.5;
chldCurrent.dHeightAtAge[3] = 104.5;
chldCurrent.dHeightAtAge[4] = 112;
etc
```

Arrays *of* objects

- Of more interest, we can have arrays of objects
- Just as an array of double might contain 750 numbers, all with the same name, accessible through a few lines of code by varying the index,
- so an array of Employee might contain 750 employees, all with the same (array) name, accessible through a few lines of code by varying the index

```
Employee[] empStaff = new Employee[750];
  :     :    :
for (iEmpNo=1; iEmpNo<=iNumStaff; iEmpNo++)
{
    CalculatePay(empStaff[iEmpNo]);
}
```

Array of objects – example

- Let’s work through an example that uses an array of objects
  - remember, arrays of objects mean we no longer need to use parallel arrays
- On the way, we will refresh our knowledge of arrays and of objects
- We’ll define a player class for some unspecified sporting club
- The array of players will be a squad, from which teams are drawn to play particular games

Think about the attributes

- What attributes do we want a player to have?
- Simplifying things a little, perhaps
  - Name
  - Phone number
  - Fees owing
  - Health issues
  - Behaviour issues
  - Games played
  - Preferred position
  - Season rating
  - Selected for next game

Think about the properties

- Remember, the attributes should all be private, and accessed by way of public properties
- Some of the properties will simply echo the attributes, while others might impose some control over the way the attributes are accessed
- Think back to the attributes. Should any of the properties be read-only or write-only? Should any of them have any other control imposed on the way they are accessed?

Think about the methods

- What methods will players require? That is, what sorts of thing should players be able to do?
- We might choose a register method, in which players sign up with the squad,
- a play game method, in which players play a game and then undergo changes in some of their properties,
- and a pay fees method

Instantiate array *and* objects

- A single player might be declared and instantiated

```
Player plrSquadMember = new Player();
```

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

```
Player[] plrSquad = new Player[21];
```

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

```
for (plrThis = 1; plrThis <= 20; plrThis++)
    {  plrSquad[plrThis] = new Player();  }
```

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

Registering a player

- To register a player we need to
  - use the next available element of the array
  - create (instantiate) a new player object
  - give it a name, a phone number, a preferred position, and health issues
  - give it default values for fees owing, games played, and season rating
  - leave behaviour issues blank
  - leave selected for next game false

Playing a game

- Once a player has played a game,
  - games played should be increased by 1
  - health issues, behaviour issues, and season rating will need updating
  - the other attributes should remain unaltered

Paying fees

- When a player pays fees
  - the amount paid should be subtracted from the fees owing
  - the other attributes should remain unaltered

Putting it all together

- The Sport squad demo puts all of this together, with a little bit more to show the array in action
- In SportSquad, note the array of players, and the two integers to keep track of it
- Note how the navigation buttons move from player to player
- Note how the data for the current player is displayed on the form
- Note when the display methods are called

Examining the code

- We've added an InputBox form, and written InputString and InputDouble methods that use the form
- However, when collecting data for a new player we illustrate a different approach
- The textboxes look normal, but they are set to ReadOnly, so the user can't write in them. But for data entry we temporarily turn off that setting, so as to accept data directly in the textboxes
- We also change the colour of the textboxes, to draw them to the user's attention – which means we have to remember the old colour so we can restore it afterwards
- We set the TabIndex of those boxes appropriately, too

Examining the code

- In Player, note the private attributes
- Note the public properties; see which of them are read-only, and what control some of them impose on the attributes
- Note the simple idea of using the informative name for the property and adding '_' for the attribute
- Note the public methods
- Note the calculation in NewRating; it looks tricky, but it's not really too hard to understand if you try
- Remember we said you should comment any non-obvious code? Look at the proportion of comment to code in this method!

A human resource system

- When we start to design a solution to a problem, one of the first tasks is to identify the ‘things’ within the problem space
- These things will then generally be written as classes
- Human Resource systems are used to manage the people within an organisation
- If you were asked to build a University Human Resource system, what ‘things’ would you expect to deal with in the problem space?
- Student, teacher, secretary, cleaner . . .

Which attributes?

- We are only interested in the attributes of the class that describe the class in the context *we* are dealing with
- For example, which of these attributes would be important in a teacher class?
  – eye color
  – height
  – qualifications
  – telephone number
  – marital status
  – courses taught

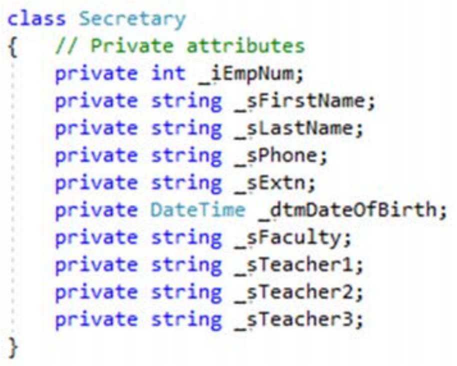
Which methods?

- As well as the attributes of a class, we need to identify its methods, the things its objects do
- For example
  - teachers
    - teach courses
    - mark assignments
  - students
    - enrol in courses
    - pay fees
    - submit assignments

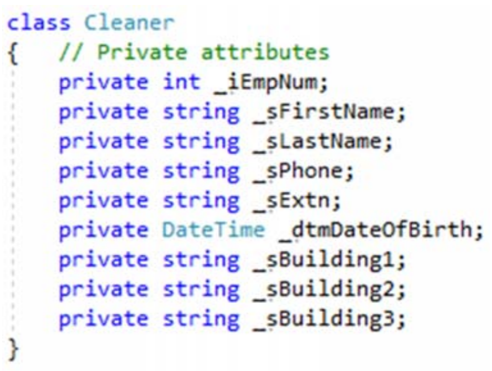
Which attributes?

- Some of the attributes we might have chosen are . . .
- Teacher – first name, last name, phone number, extension, date of birth, highest qualification, course 1, course 2, course 3
- Cleaner – first name, last name, phone number, extension, date of birth, building1, building2, building3
- Secretary – first name, last name, phone number, extension, date of birth, faculty, teacher1, teacher2, teacher3

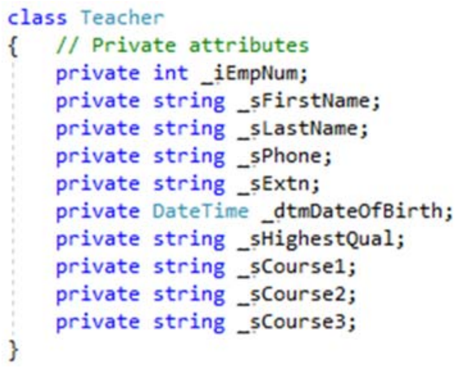
Common attributes



```
class Secretary
{   // Private attributes
    private int _iEmpNum;
    private string _sFirstName;
    private string _sLastName;
    private string _sPhone;
    private string _sExtn;
    private DateTime _dtmDateOfBirth;
    private string _sFaculty;
    private string _sTeacher1;
    private string _sTeacher2;
    private string _sTeacher3;
}
```



```
class Cleaner
{   // Private attributes
    private int _iEmpNum;
    private string _sFirstName;
    private string _sLastName;
    private string _sPhone;
    private string _sExtn;
    private DateTime _dtmDateOfBirth;
    private string _sBuilding1;
    private string _sBuilding2;
    private string _sBuilding3;
}
```



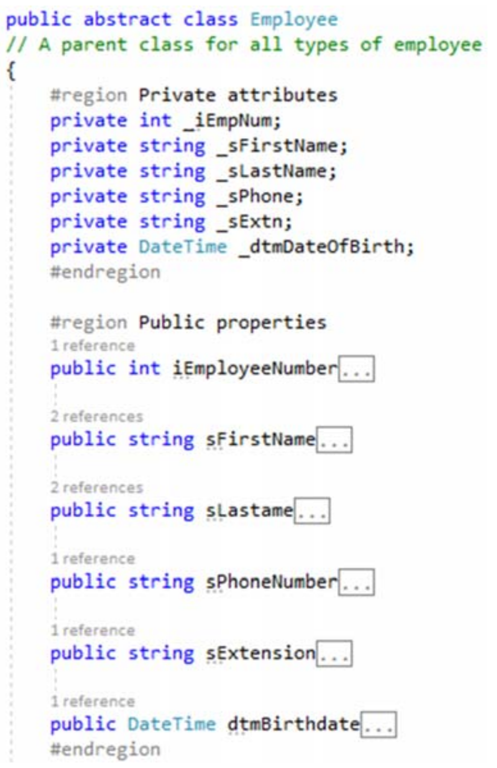
```
class Teacher
{   // Private attributes
    private int _iEmpNum;
    private string _sFirstName;
    private string _sLastName;
    private string _sPhone;
    private string _sExtn;
    private DateTime _dtmDateOfBirth;
    private string _sHighestQual;
    private string _sCourse1;
    private string _sCourse2;
    private string _sCourse3;
}
```

Many of the attributes are common to each of these classes:

- _iEmpNum
- _sFirstName
- _sLastName
- _sPhone
- _sExtn
- _dtmDateOfBirth

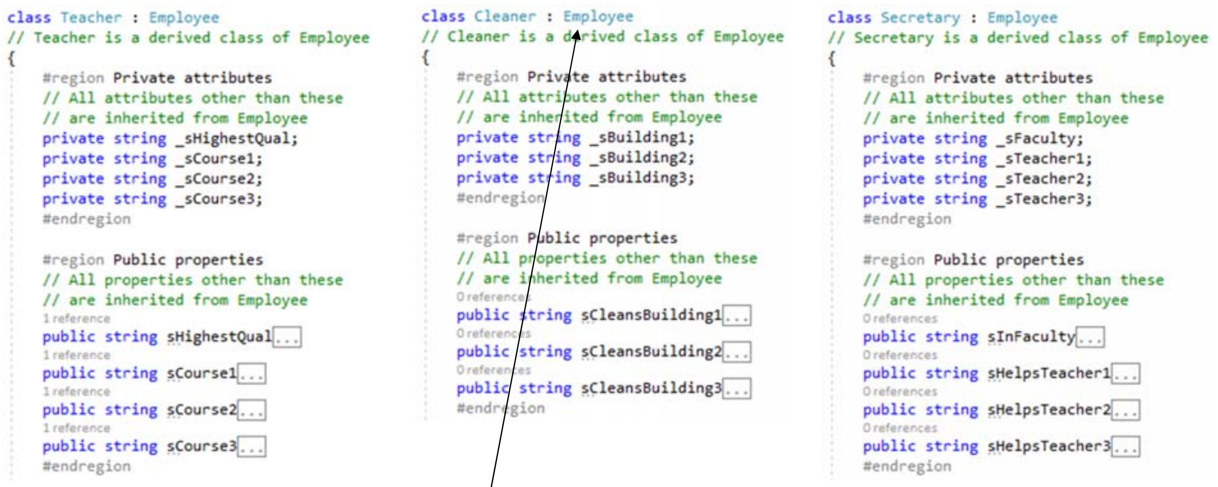
Inheritance

- Rather than repeat the common attributes for each class, why not take them out and reuse them? This is the idea behind <u>inheritance</u>.
- We can create an Employee class with the common attributes / properties
- An *abstract* class is one that can't have instances, though its subclasses can



```
public abstract class Employee
// A parent class for all types of employee
{
    #region Private attributes
    private int _iEmpNum;
    private string _sFirstName;
    private string _sLastName;
    private string _sPhone;
    private string _sExtn;
    private DateTime _dtmDateOfBirth;
    #endregion

    #region Public properties
    1 reference
    public int iEmployeeNumber...

    2 references
    public string sFirstName...

    2 references
    public string sLastame...

    1 reference
    public string sPhoneNumber...

    1 reference
    public string sExtension...

    1 reference
    public DateTime dtmBirthdate...
    #endregion
```

Inheritance



```
class Teacher : Employee
// Teacher is a derived class of Employee
{
    #region Private attributes
    // All attributes other than these
    // are inherited from Employee
    private string _sHighestQual;
    private string _sCourse1;
    private string _sCourse2;
    private string _sCourse3;
    #endregion

    #region Public properties
    // All properties other than these
    // are inherited from Employee
    1 reference
    public string sHighestQual[...]
    1 reference
    public string sCourse1[...]
    1 reference
    public string sCourse2[...]
    1 reference
    public string sCourse3[...]
    #endregion
```

```
class Cleaner : Employee
// Cleaner is a derived class of Employee
{
    #region Private attributes
    // All attributes other than these
    // are inherited from Employee
    private string _sBuilding1;
    private string _sBuilding2;
    private string _sBuilding3;
    #endregion

    #region Public properties
    // All properties other than these
    // are inherited from Employee
    0 references
    public string sCleansBuilding1[...]
    0 references
    public string sCleansBuilding2[...]
    0 references
    public string sCleansBuilding3[...]
    #endregion
```

```
class Secretary : Employee
// Secretary is a derived class of Employee
{
    #region Private attributes
    // All attributes other than these
    // are inherited from Employee
    private string _sFaculty;
    private string _sTeacher1;
    private string _sTeacher2;
    private string _sTeacher3;
    #endregion

    #region Public properties
    // All properties other than these
    // are inherited from Employee
    0 references
    public string sInFaculty[...]
    0 references
    public string sHelpsTeacher1[...]
    0 references
    public string sHelpsTeacher2[...]
    0 references
    public string sHelpsTeacher3[...]
    #endregion
```

Notice the colon and superclass name. Each of these staff members is a 'derived' class that inherits the attributes, properties, and methods of the Employee class

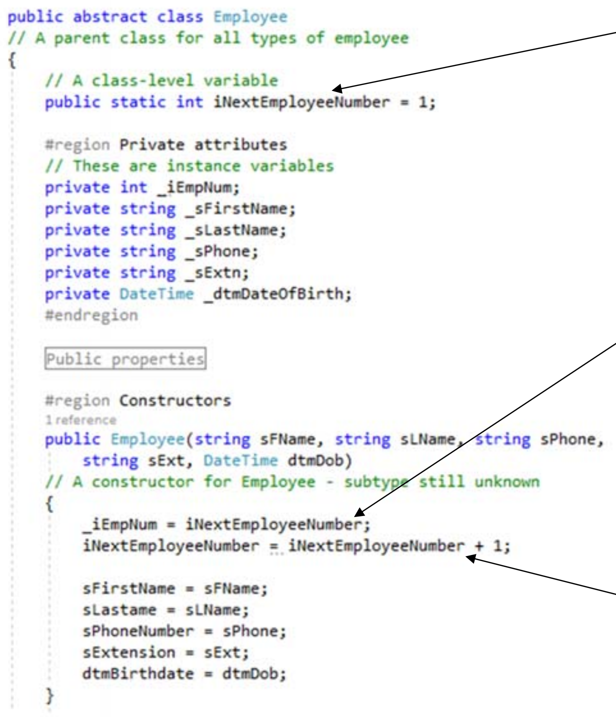
Private, public, and protected

- Let's have a method in Employee called sGetName, which appends last name, comma, space, and first name
- If it's *private*, not even its derived classes can see it; base.sGetName() means 'the sGetName method of the parent class of this class', but this fails
- If it's *public*, the derived classes can see it, but so can the rest of the program, which might not be appropriate
- So we make it *protected*, which means the derived classes can see it, but the rest of the program can't
- Private, public, and protected (and a few more) are called *access modifiers*, because they control how certain parts of the program can access other parts

Shared attributes

- Imagine that we want to create a unique Employee number; we need to know which number to use next
- We could create a number in a database and use that, but we don't yet know how to work with databases
- We could have the UI (the form) maintain a value and pass this to the class; but this doesn't seem right
- We'd like to have the class create its own value
- How can the class keep track of which number is next?
- A *shared attribute* or *class attribute*, marked with the word 'static', is a single attribute that belongs to the class as a whole, rather than each instance having its own copy

A shared sequence number



```
public abstract class Employee
// A parent class for all types of employee
{
    // A class-level variable
    public static int iNextEmployeeNumber = 1;

    #region Private attributes
    // These are instance variables
    private int _iEmpNum;
    private string _sFirstName;
    private string _sLastName;
    private string _sPhone;
    private string _sExtn;
    private DateTime _dtmDateOfBirth;
    #endregion

    Public properties

    #region Constructors
    1 reference
    public Employee(string sFName, string sLName, string sPhone,
        string sExt, DateTime dtmDob)
    // A constructor for Employee - subtype still unknown
    {
        _iEmpNum = iNextEmployeeNumber;
        iNextEmployeeNumber = iNextEmployeeNumber + 1;

        sFirstName = sFName;
        sLastame = sLName;
        sPhoneNumber = sPhone;
        sExtension = sExt;
        dtmBirthdate = dtmDob;
    }
```

`static` means that iNextEmployeeNumber belongs to the whole Employee Class

We assign a new employee's number to be the current value of iNextEmployeeNumber

We increment iNextEmployeeNumber after each use, so it always contains the next employee number to use

Static means class-level

- The previous example doesn’t really show the big difference in usage between static variables and instance variables
- This is because the variable is used within the class.
- Used outside the class . . .
- an instance variable is described by <instanceName>.<variableName>, eg empNewStaffMember.iEmployeeNumber
- while a static variable is described by <className>.<variableName>, eg Employee.iNextEmployeeNumber

All members can be static

- So far we’ve just seen a static variable
- Properties and methods can also be static
- We have already seen properties that belong to classes, not their instances, eg Color.Black, Pens.Blue, Math.PI; these are defined as static
- We have also seen methods that belongs to classes, not their instances, eg Math.Sqrt(); these are also defined as static
- If you want a variable, property, or method to be associated with the class rather than each instance of the class, make it static

Inheritance – summary

- In general, if several classes have some common attributes, properties and methods, create a class of these attributes, properties and methods that the other classes can inherit
- The Human Resources demo illustrates inheritance with the human resource system. It doesn’t *do* much, but it has code for all the concepts we’ve discussed here, and some more besides
- It does display the employee number every time a new Teacher is created, to show that the static variable works
- It also displays the next employee number, to show how it’s called

Features of OO programming

- The buzz words of OO (object-oriented) programming include data hiding, inheritance, polymorphism and encapsulation
- *Data hiding* means using *private* to hide data from other entities, then permitting controlled access by way of public properties
- *Inheritance* means gathering and reusing common attributes, properties, and methods
- *Encapsulation* means keeping together the data and the methods that act on that data; we do this when we create a class
- That leaves *polymorphism* – objects that can take many forms

Inheritance again

- The Employee class has three different derived classes, which have some common properties and some distinct properties
- If you wanted a form like the SportSquad one, to display all employees in turn, what fields would you have on the form?
- Would they all be there, and some of them left blank (eg courses taught would be blank for a cleaner)?
- Or might there be a better way?

Polymorphism

- In a Human Resources program like the SportSquad one we would have an array of Employee
- It has to be an array of Employee, because some of them are Cleaner, some Secretary, and some Teacher, and yet we want them all in the same array
- So each element of the array is an Employee; but when it's assigned a value, it becomes a Cleaner, a Secretary, or a Teacher
- The array elements are *polymorphic*: they can take on different forms, depending on what value is assigned to them
- In the demo, empNewStaffMember is declared as Employee but then instantiated as Teacher

Which derived class?

- If we want the form to appear different for each subclass, when we display an element of the array we need to know which subclass that element is
- One possible approach is to use the GetType() method of the element. This returns its type, so empStaffMember[i].GetType() will return Lec11Demo.Cleaner, Lec11Demo.Secretary, or Lec11Demo.Teacher.

We can’t use classes in expressions

- Unfortunately, we can’t say

  ```
  if (empStaffMember[i].GetType() ==
     Lec11Demo.Cleaner)
  ```

  because the type can’t be used in an expression.
- But we can say

  ```
  if (empStaffMember[i].GetType().ToString()
     == "Lec11Demo.Cleaner")
  ```

- So this is how we can tell which derived class a particular element is . . .
- . . . and then go on to adjust the form (and the program) accordingly

Object-oriented programming

- This course teaches using object-oriented programming
- We haven't spent much time on creating classes
- However, almost everything we've done involves declaring and using objects – of classes that already exist in the language
- The only real difference is that now we know how to define our own classes and declare objects of those classes
- And that's pretty much it for the course
- Next week we'll do some revision and exam preparation