Name:	

Course: Introduction to .NET
Title: Objects
Instructor: Bill Buchanan

NAPIER UNIVERSITY
EDINBURGH SCOTLAND

8 Objects

8.1 Introduction

This module covers the key elements of arrays, indexers and collections. It covers:

- Classes and objects. This revises the material covered in Module 1.
- Using encapsulation.
- Defining Object-Oriented Systems.
- Inheritance.
- Abstract classes.
- Sealed classes.
- Overriding classes.

8.2 Classes and objects

We live in a world full of objects; so object-oriented programming is a natural technique in developing programs. For example, we have an object called a cup, and each cup has a number of **properties**, such as its colour, its shape, its size, and so on. It is efficient for us to identify it as a cup, as we know that cups should be able to hold liquid, and we will place our cup beside all the other cups that we have. If we were a cup designer then we could list all the possible properties of a cup, and for each design, we could set the properties of the cup. Of course, some of the properties might not actually be used, but for a general-purpose design, we would specify every property that a cup might have. For example, in a simple case we could have the following properties:

Properties	**Cup 1**	**Cup 2**	**Cup3**
Shape (Standard/Square/Mug)	Standard	Square	Mug
Colour (Red/Blue/Green)	Blue	Red	Green
Size (Small/Medium/Large)	Small	Large	Small
Transparency (0 to 100%)	100%	50%	25%
Handle type (Small/Large)	Small	Small	Large

Thus, we have three choices of shape (square, standard or mug), three choices of colour (red, blue or green), three choices in size (small, medium or large) and two choices of handle type (small or large). In addition, we can also choose a level of transparency of the cup from 0 to 100% (in integer steps). In object-oriented program, the collection of properties is known as a **class**. Thus, we could have a class for our cup which encapsulates all the design parameters for our cup. The instance of our class, such as Cup 1, Cup 2 and Cup 3, are known as **objects**. We can create many objects from our class. Along with this, there are certain things that we want to do with the cup, such as picking it up, painting it, or even dropping it. In object-orientation, these are known as methods, and are the functions that can be allowed to operate on our objects.

Program **1** shows an object-oriented example of the cup, where a class named `Cup` is created, of which an instance is named `cup`. A full description on this program will be discussed in a later module. It uses variables, such as `Shape`, `Colour`, and `Size` to define the property of an object.

Program 8.1: Program1_1_ClassSimpleCup

```
using System;

namespace ConsoleApplication2
{
   public class Cup
   {
      public string Shape;
      public string Colour;
      public string Size;
      public int Transparency;
      public string Handle;

      public void DisplayCup()
      {
         System.Console.WriteLine("Cup is {0}, {1}", Colour, Handle);
      }
   }
   class Class1
   {
      static void Main(string[] args)
      {
         Cup cup = new Cup();
         cup.Colour = "Red";        cup.Handle = "Small";
         cup.DisplayCup();
         cup.Colour = "Green";      cup.Handle = "Small";
         cup.DisplayCup();
         System.Console.ReadLine();
      }
   }
}
```

Sample Run

```
Cup is Red, Small
Cup is Green, Small
```

In the following example, we create a class named `Circuit`, of which we create a new instance of it named `cir`. The class then has two methods, named `Serial` and `Parallel`.

Program 8.2:

```
using System;

namespace ConsoleApplication2
{
   public class Circuit
   {
      public string name;

      public double Parallel(double r1, double r2)
      {
         return((r1*r2)/(r1+r2));
      }
      public double Series(double r1, double r2)
      {
         return(r1+r2);
      }
   }

   class Class1
   {
      static void Main(string[] args)
      {
         double v1=100,v2=100;
         double res;

         Circuit cir = new Circuit();

         cir.name="Circuit 1";
         res=cir.Parallel(v1,v2);
         System.Console.WriteLine("[{0}] Parallel resistance is {1} ohms",
               cir.name,res);

         cir.name="Circuit 2";
         res=cir.Series(v1,v2);
         System.Console.WriteLine("[{0}] Series resistance is {1} ohms ",
               cir.name,res);

         System.Console.ReadLine();
      }
   }
}
```

Sample Run

```
[Circuit 1] Parallel resistance is 50 ohms
[Circuit 2] Series resistance is 200 ohms
```

In this case, we have used a single object (`cir`). Of course, we could have created two objects, with:

```
Circuit cir1 = new Circuit();
Circuit cir2 = new Circuit();

cir1.name="Circuit 1";  res1=cir1.Parallel(v1,v2);
cir2.name="Circuit 2";  res2=cir.Series(v1,v2);
```

Finally, for this section, Program 8.3 shows an example of a complex number class, where a complex number object is created (`r`), which is made up of two components (`r.real` and `r.imag`). The class defines two methods: `mag()` and `angle()`, which calculate the magnitude and the angle of the complex number, using:

$$z = x + \mathrm{j}y$$

$$|z| = \sqrt{x^2 + y^2}$$

$$\langle z \rangle = \tan^{-1}\left(\frac{y}{x}\right)$$

Program 8.3:

```
using System;
namespace ConsoleApplication2
{
   public class Complex
   {
      public double real;
      public double imag;

      public double mag()
      {
         return (Math.Sqrt(real*real+imag*imag));
      }
      public double angle()
      {
         return (Math.Atan(imag/real)*180/Math.PI);
      }
   }
   class Class1
   {
      static void Main(string[] args)
      {
         string str;
         double mag,angle;
         Complex r = new Complex();

         System.Console.Write("Enter real value >>");
         str=System.Console.ReadLine();
         r.real = Convert.ToInt32(str);

         System.Console.Write("Enter imag value >>");
         str=System.Console.ReadLine();
         r.imag = Convert.ToInt32(str);

         mag=r.mag();
         angle=r.angle();

         System.Console.WriteLine("Mag is {0} and angle is {1}",mag,angle);
         System.Console.ReadLine();
      }
   }
}
```

Sample Run

```
Enter real value >> 3
Enter imag value >> 4
Mag is 5 and angle is 53.130102354156
```

8.3 Using encapsulation

Properties are useful techniques in defined the state of a class. For example, a car might have the following properties:

- **Make**. This could be Ford, Vauxhall, Nissan or Toyota.
- **Type**. This could be Vectra, Astra, or Mondeo.
- **Colour**. This could be colours such as Red, Green or Blue.
- **Country of Manufacture**. This could be countries such as UK, Germany or France.

Normally in object-oriented programs, the variable members of the class are not make public, thus we can defined a property which can be used to set these variables. Program 8.4 shows an example where the variables make, type, colour and country are made private, but properties are defined for these, in order to set them.

Program 8.4:

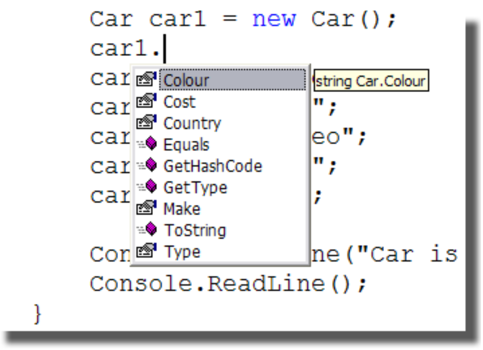
```
using System;

namespace sample01
{
   public class Car
   {
      private string colour;
      private string type;
      private string make;
      private string country;
      private double cost;

      public string Colour
      {
         get { return colour; }
         set { colour=value; }
      }

      public string Type
      {
         get { return type; }
         set { type=value; }
      }
      public string Country
      {
         get { return country; }
         set { country=value; }
      }
      public string Make
      {
         get { return make; }
         set { make=value; }
      }
      public double Cost
      {
         get { return cost; }
         set { cost=value; }
      }
```

```
    }
    class Class1
    {
       static void Main(string[] args)
       {
          Car car1 = new Car();

          car1.Colour="Red";
          car1.Make="Ford";
          car1.Type="Mondeo";
          car1.Colour="UK";
          car1.Cost=15000;

          Console.WriteLine("Car is a " + car1.Make + " " + car1.Type);
          Console.ReadLine();
       }
    }
}
```

Sample Run

```
Car is a Ford Mondeo
```

It can be seen that the properties defines a set and get definitions. These are named the `get` and `set` **accessors**, and are similar to methods. The `get` accessor reads the value of the field, while the `set` accessor reads its value. In most cases, the .NET environment senses the use of the property and shows the options for it as the user types the statement, such as:



```
Car car1 = new Car();
car1.
car
car
car
car
car
Cor                    ne("Car is
Console.ReadLine();
}
```

The value part of the set in Program 8.4 defines that the value from the property which is set is taken from. For example:

```
public double Cost
{
   get { return cost; }
   set { cost=value; }
}
```

The value of cost will take the value from `object.Cost`.

8.3.1 Read-only fields

A property with only a `get` accessor is read-only, whereas a property with only a `set` is write-only. If it has both a `set` and `get` assess it is a read/write property. It is important to encapsulate the class correctly so that the objects cannot be accessed in an incorrect way. For example if we change the Cost property to make it read-only:

```
public double Cost
{
   get { return cost; }
}
```

and try to use it:

```
car1.Cost=15000;
```

The builder will return an error of:

```
Property or indexer 'sample01.Car.Cost' cannot be assigned to -- it is read
only
```

But, we could use it, such as:

```
   Console.WriteLine("The cost is " + car1.Cost);
```

8.4 Defining Object-Oriented Systems

Classes exist within a structure and interlink with others. For example if we think animals we can group them animals into a generalisation, which has a common behaviour and characteristics. We could then sub-divide mammals up into different classifications, such cats, dog and horses (as illustrated in Figure 8.1). From there we can further subdivide until we get to actual species. Thus we generalise at the top of the hierarchy, and specialise as we move down it. In object-orient techniques, we create a hierarchy such as this, which starts with generalised class, which then future sub-divide into specialisations. In C# the top level is System namespace, which has certain splits of into other namespaces, such as `Windows`, `IO`, and `Drawing`, as illustrated in Figure 8.2. Below `System.Windows` is `System.Windows.Forms`, which has a number of classes assigned to it. These include:

- Form. To add a form.
- TextBox. Which creates a textbox.
- ComboBox. Which creats a combobox.
- ScrollBar. Which creates a scrollbar.

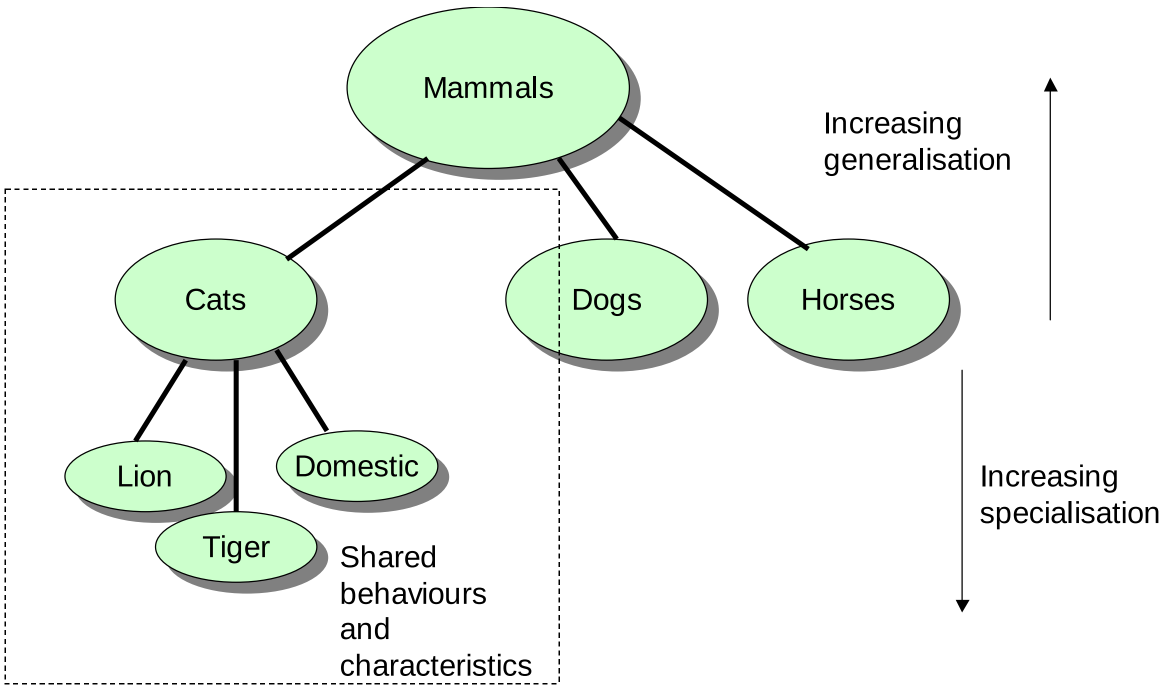

Mammals
Increasing generalisation
Cats
Dogs
Horses
Lion
Domestic
Tiger
Shared behaviours and characteristics
Increasing specialisation

Figure 8.1: Animal classifications

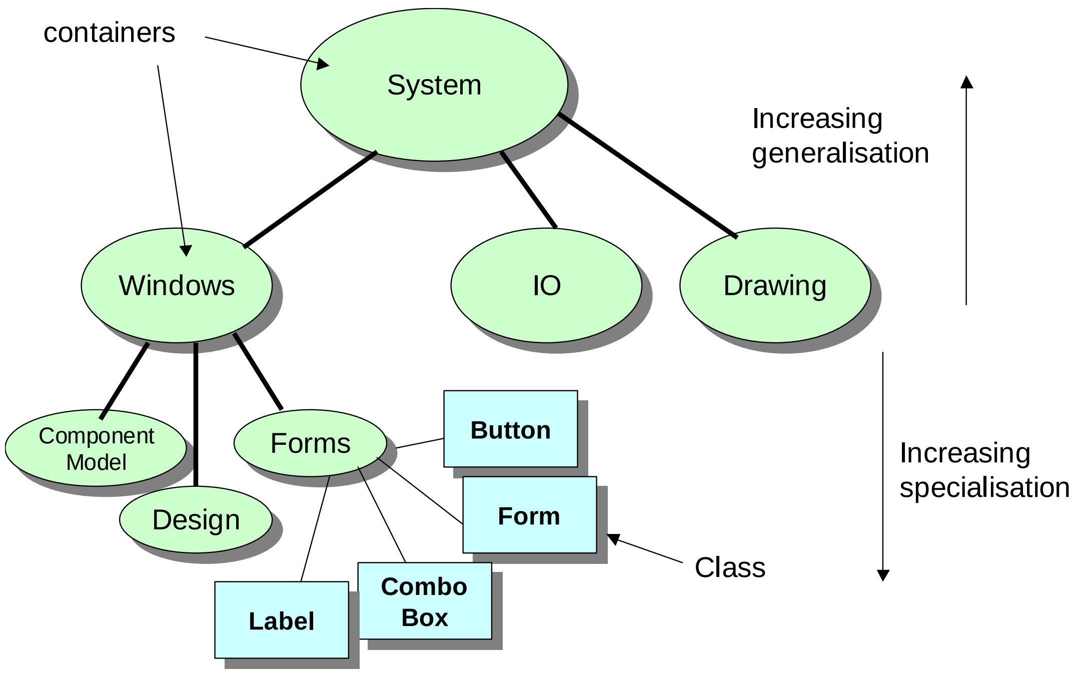


Figure 8.2: Object hierarchy in C#

Thus, we can say that a Button is a kind of Forms (excuse the poor use of English, though). Program 8.5 shown an example program with a form, a textbox and a button. It can be seen that the form is defined as a class with:

Derived class

Base class

```
public class Form1 : System.Windows.Forms.Form
```

which means that our form derives from `System.Windows.Forms.Form`. The button and the text box are defined with:

```
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.TextBox textBox1;
```

Which means they derive from `System.Windows.Forms.Button` and `System.Windows.Forms.TextBox`, respectively.

Program 8.5:

```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Base class

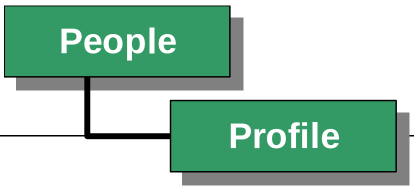
```
namespace WindowsApplication3
{
   public class Form1 : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Button button1;
      private System.Windows.Forms.TextBox textBox1;

      public Form1()    {       InitializeComponent();    }

      private void InitializeComponent()
      {
         this.button1 = new System.Windows.Forms.Button();
         this.textBox1 = new System.Windows.Forms.TextBox();

         this.button1.Location = new System.Drawing.Point(152, 192);
         this.button1.Name = "button1";
         this.button1.TabIndex = 0;
         this.button1.Text = "button1";
         this.textBox1.Location = new System.Drawing.Point(80, 64);
         this.textBox1.Name = "textBox1";
         this.textBox1.TabIndex = 1;
         this.textBox1.Text = "textBox1";
         this.Controls.Add(this.textBox1);
         this.Controls.Add(this.button1);
         this.Name = "Form1";
         this.Text = "Form1";
         this.Load += new System.EventHandler(this.Form1_Load);
         this.ResumeLayout(false);
      }

      static void Main()
      {
         Application.Run(new For m1());
      }
      private void Form1_Load(object sender, System.EventArgs e)
      {
      // event for the loading of the form
      }
   }
}
```

8.4.1 Inheritance

Inheritance is a key factor in C#, where derived classes inherit all the members of the base classes. In Program 8.6 a class named `People` is created. This has a property of name, and a method of `ToLowerCase()`. Next a `Profile` class is created, which derives from `People`. Profile thus inherits all the members of `People`.

Program 8.6:

```
using System;
namespace ConsoleApplication1
{
   class People
   {
      private string name;
```



```
      public string Name {
         get {return name;}
         set {name=value;}
      }
      public void ToLowerCase()
      {
         name=name.ToLower();
      }
   }

   class Profile : People
   { // Inherit from People
      public void Display()
      {
         Console.WriteLine("Name: "+Name);
      }
   }

   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();

         p1.Name="Fred";

         p1.ToLowerCase();
         p1.Display();

         System.Console.ReadLine();
      }
   }
}
```

Method inherited from People

Sample Run

```
Name: fred
```

If we now change the class to:

```
   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();
         Profile p2 = new Profile();

         p1.Name="Fred";
         p2.Name="Bert";

         p1.ToLowerCase();
         p1.Display();
         p2.Display();
         System.Console.ReadLine();
      }
   }
```

Then the output will become:

```
Name: fred
Name: Bert
```

Where the first name (Fred) has had the `ToLowerCase()` method applied to it, but the second one (Bert) has not.

8.5 Sealed classes

The sealed modifier is used to bar classes from deriving its members, and is used to prevent deviation from the original purposes of a class. For example, if in Program 8.6 we change the People class to:

```
  sealed class People
  {
    private string name;

    public string Name {
      get {return name;}
      set {name=value;}
    }
    public void ToLowerCase()
    {
      name=name.ToLower();
    }
  }
```

When the program is now built an error messages is displayed:

```
'ConsoleApplication1.Profile' : cannot inherit from sealed class
'ConsoleApplication1.People'
```

8.6 Overriding classes

Sometimes a developer might want to implement their own methods. This could be to enhance the operation of the method, or to refine it for a certain application or system. For this the override keyword is used to identify that a method overrides a method from the base class. An example in Program 8.7 shows that the overriding class is defined with:

```
    public override void ToLowerCase()
    {
      Console.WriteLine("Method has been overwritten");
```

```
     }
```

and the virtual keyword is used in the definition of the method in the base class, such as:

```
     public virtual void ToLowerCase()
     {
        name=name.ToLower();
     }
```

Program 8.7:

```
using System;
namespace ConsoleApplication1
{
   class People
   {
      private string name;

      public string Name
      {
         get {return name;}
         set {name=value;}
      }
      public virtual void ToLowerCase()
      {
         name=name.ToLower();
      }
   }

   class Profile : People
   { // Inherit from People
      public void Display()
      {
         Console.WriteLine("Name: "+Name);
      }
      public override void ToLowerCase()
      {
         Console.WriteLine("Method has been overwritten");
      }
   }

   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();

         p1.Name="Fred";

         p1.ToLowerCase();
         p1.Display();

         System.Console.ReadLine();
      }
   }
}
```

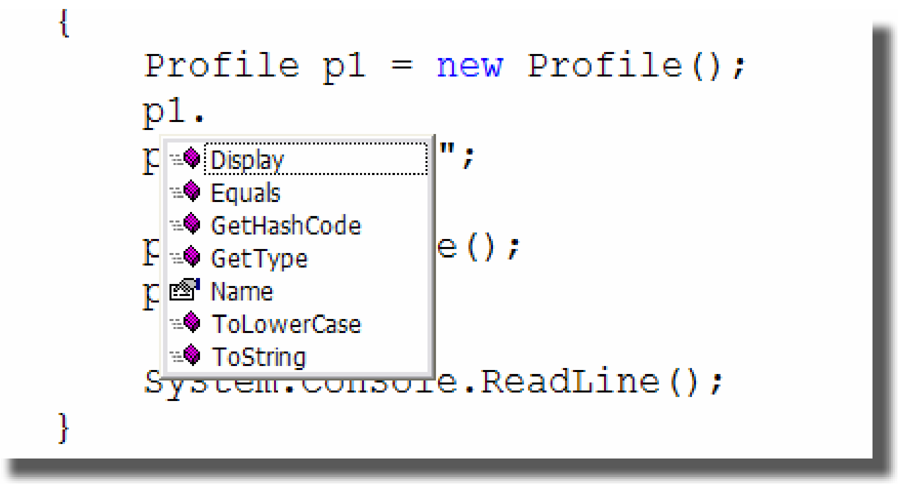
Sample Run

```
Method has been overwritten
Name: Fred
```

Every class derives from the Object, of which the main methods are:

- `Equals()`. Determines if two objects are the same.
- `Finalize()`. Cleans up the object (the destructor).
- `GetHashCode()`. Allows an object to define its hash code.
- `GetType()`. Determine the type of an object.
- `MemberwiseClone()`. Create a copy of the object.
- `ReferenceEquals()`. Determines whether two objects are of the same instance.
- `ToString()`. Convert an object to a string.

It can be seen from Figure 8.3 that some of these methods have been derive from the main object model. As an example, we can override these methods for any object that is created.



```
{
    Profile p1 = new Profile();
    p1.
    p                ";
    p                e();
    p
    System.Console.ReadLine();
}
```

Figure 8.3: Example of object method

Program 8.8:

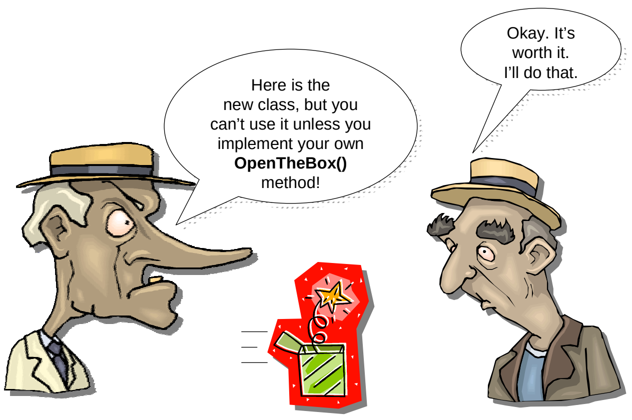
```
using System;
namespace ConsoleApplication1
{
   class People
   {
      private string name;

      public string Name
      {
         get {return name;}
         set {name=value;}
      }
      public virtual void ToLowerCase()
      {
         name=name.ToLower();
      }
   }

   class Profile : People
   { // Inherit from People
      public void Display()
      {
         Console.WriteLine("Name: "+Name);
      }

      public override string ToString()
      {
         return("Cannot implement this method");
      }
   }

   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();
         p1.Name="Fred";

         string str=p1.ToString();
         Console.WriteLine(str);
         System.Console.ReadLine();
      }
   }
}
```

Sample Run

```
Method has been overwritten
Name: Fred
```

8.7 Abstract classes

There are times in programming when the developer of a class wants to force the classes which derive the class to write their own methods. This might be where a system could use different types of graphics displays, and that the developer should write their own implementation of a `DrawWindow()` method. An abstract class can be used for this,


Here is the new class, but you can't use it unless you implement your own **OpenTheBox()** method!
Okay. It's worth it. I'll do that.

where the class contains an abstract definition of the format of the At times a new class must be forced to create a new method. In C# it is not possible to derive a class if there is not an implementation of an abstract method contained in it.

For example if we have an abstract class for a Car:

```
public abstract class Car
{
      public abstract void ShowColour();
      public abstract void ShowType();
}
```

This defines that the deriving class must implement these methods. As it is an abstract class, there is no implementation as this is up to the derived class to implement it. Using the previous example, we can force the derived class (Profile) to implement the `DisplayLower()` method with the addition of the line:

```
public abstract void DisplayLower();
```

in the `People` class.

Program 8.9:

```
using System;
namespace ConsoleApplication1
{
   class People
   {
      private string name;

      public string Name
      {
         get {return name;}
         set {name=value;}
      }
      public void ToLowerCase()
      {
         name=name.ToLower();
      }
      public abstract void DisplayLower();
   }

   class Profile : People
   { // Inherit from People
      public void Display()
      {
         Console.WriteLine("Name: "+Name);
      }
   }

   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();

         p1.Name="Fred";

         p1.ToLowerCase();
         p1.Display();

         System.Console.ReadLine();
      }
   }
}
```

The builder will give the following error:

```
'ConsoleApplication1.Profile' does not implement inherited abstract
member 'ConsoleApplication1.People.DisplayLower()'
```

To overcome this we would have to add the following method:

```
      public override void DisplayLower()
      {
            Console.WriteLine("Name: "+Name.ToUpper());
      }
```

which will implement the required method. Program 8.10 shows the full implementation of the program.

Program 8.10:

```
using System;
namespace ConsoleApplication1
{
   abstract class  People
   {
      private string name;

      public string Name
      {
         get {return name;}
         set {name=value;}
      }
      public void ToLowerCase()
      {
         name=name.ToLower();
      }
      abstract public void DisplayLower();
   }

   class Profile : People
   { // Inherit from People
      public void Display()
      {
         Console.WriteLine("Name: "+Name);
      }
      public override void DisplayLower()
      {
            Console.WriteLine("Name: "+Name.ToUpper());
      }
   }

   class Test
   {
      static void Main()
      {
         Profile p1 = new Profile();

         p1.Name="Fred";

         p1.ToLowerCase();
         p1.Display();
         p1.DisplayLower();

         System.Console.ReadLine();
      }
   }
}
```

Sample Run

```
Name: fred
Name: FRED
```

Obviously, the developer does not actually need to put anything in the implementation of the code, if they do not want to. For this, they can create an empty method.

```
public override void DisplayLower()    { }
```

8.8 Possible Solutions

```
using System;

namespace ConsoleApplication6
{
   public class InstrumentMeasurement
   {
      string tag, range, time;
      double voltage, power;

      public string Tag { set { tag=value; } }
      public string Range { set { range=value; } }
      public string Time { set { time=value; } }
      public double Voltage { set { voltage=value; } }
      public double Power { set { power=value; } }
      public void Display()
      {
         Console.WriteLine("Tag: " + tag + " Voltage " + voltage);
      }
   }

   class class1
   {

      static void Main(string[] args)
      {
         InstrumentMeasurement Data1 = new InstrumentMeasurement();
         Data1.Tag="AB12345"; Data1.Voltage=5.6; Data1.Power=1.2;
         Data1.Range="mW"; Data1.Time="01/07/2004 06:17:55";
         Data1.Display();

         InstrumentMeasurement Data2 = new InstrumentMeasurement();
         Data2.Tag="AB12345"; Data2.Voltage=5.3; Data2.Power=1.21;
         Data2.Range="mW"; Data2.Time="01/07/2004 06:17:56";
         Data2.Display();
         Console.ReadLine();
      }
   }
}
```

```
using System;
using System.Collections;

namespace ConsoleApplication6
{
   public class InstrumentMeasurement
   {
      string tag, range, time;
      double voltage, power;

      public string Tag { set { tag=value; } }
```

```
        public string Range { set { range=value; } }
        public string Time { set { time=value; } }
        public double Voltage { set { voltage=value; } }
        public double Power { set { power=value; } }
        public void Display()
        {
            Console.WriteLine("Tag: " + tag + " Voltage " + voltage);
        }
    }

    class class1
    {

        static void Main(string[] args)
        {
            ArrayList a1 = new ArrayList();
            InstrumentMeasurement Data1 = new InstrumentMeasurement();
            Data1.Tag="AB12345"; Data1.Voltage=5.6; Data1.Power=1.2;
            Data1.Range="mW"; Data1.Time="01/07/2004 06:17:55";


            a1.Add(Data1);

            InstrumentMeasurement Data2 = new InstrumentMeasurement();
            Data2.Tag="AB12345"; Data2.Voltage=5.6; Data2.Power=1.2;
            Data2.Range="mW"; Data2.Time="01/07/2004 06:17:55";

            a1.Add(Data2);

            for (int i=0;i<a1.Count;i++)
            {
                InstrumentMeasurement obj = (InstrumentMeasurement) a1[i];
                obj.Display();
            }

            Console.ReadLine();
        }
    }
}



            int counter=0;
            for (int i=0;i<a1.Count;i++)
            {
                InstrumentMeasurement obj = (InstrumentMeasurement) a1[i];
                if (obj.Power>0.001) counter++;
                obj.Display();
            }


//
using System;
using System.Collections;

namespace ConsoleApplication6
{
    public class InstrumentMeasurement
    {
        string tag, range, time;
        double voltage, power;

        public string Tag { set { tag=value; } }
        public string Range { set { range=value; } }
        public string Time { set { time=value; } }
        public double Voltage { set { voltage=value; }get { return voltage; }  }
        public double Power { set { power=value; } get { return power; }  }
        public void Display()
        {
            Console.WriteLine("Tag: " + tag + " Voltage " + voltage);
        }
    }
    class NewInstrumentMeasurement: InstrumentMeasurement
    {
        public bool IsVoltageLarger(double val)
        {
            if (Voltage>5) return (true);
            else return (false);
        }
    }


    class class1
    {

        static void Main(string[] args)
        {
            ArrayList a1 = new ArrayList();
            NewInstrumentMeasurement Data1 = new NewInstrumentMeasurement();
            Data1.Tag="AB12345"; Data1.Voltage=5.6; Data1.Power=1.2;
            Data1.Range="mW"; Data1.Time="01/07/2004 06:17:55";


            a1.Add(Data1);

            NewInstrumentMeasurement Data2 = new NewInstrumentMeasurement();
            Data2.Tag="AB12345"; Data2.Voltage=4.9; Data2.Power=1.2;
            Data2.Range="mW"; Data2.Time="01/07/2004 06:17:55";

            a1.Add(Data2);
            int counter=0;
            for (int i=0;i<a1.Count;i++)
            {
                NewInstrumentMeasurement obj = (NewInstrumentMeasurement) a1[i];
                if (obj.IsVoltageLarger(5)==true) counter++;
                obj.Display();
            }
            Console.WriteLine("Number greater than 5 is " + counter);

            Console.ReadLine();
        }
    }
}
```