ICT365

Software Development Frameworks

Dr Afaq Shah

Slides: Dr Giles Oatley

Murdoch
UNIVERSITY

Object-Oriented Programming (C#)

Murdoch
UNIVERSITY

In this Topic

- OOP techniques:
  - Encapsulation, information hiding,
  - modularity, and
  - polymorphism
- Understand how a C# program is organised both logically and physically
- Declare and use namespaces
- Properties

Additional notes: Exceptions

- Exception mechanism
- Catching and processing exceptions
- Throwing an exception

Object-Oriented Programming

- Object-oriented programming (OOP) - uses objects to design programs
- In OOP, a program is viewed as a collection of objects
- Each object has a certain number of attributes and a set of behaviours in the form of methods
- An object can “send” another object a “message”

Object-Oriented Programming

The first object passes the control to the second object.

The second object “receives” the “message”

Once the execution finished, the control returns to the original object.

OOP is different from procedural programming.

Features of Object Oriented Systems

OOP uses many techniques that are supported by any object-oriented language:

encapsulation (by using classes)

information hiding (by using access specifiers and properties)

modularity (by using classes hierarchy, namespaces, compilation units, etc)

polymorphism (using the same name for multiple methods with different signatures in the same class)

inheritance (using subclassing, virtual methods, method overriding, interfaces, etc)

Organisation of C# Programs

Logically, a C# program consists of a collection of classes (and structs).

organised in a hierarchy of *namespaces*.

Some of these classes are coded by you.

the standard .NET Framework Class Library.

use classes provided by a third party.

Namespaces are used to group these classes logically - rooted at System.Object.

Organisation of C# Programs

Physically, these classes are stored in a number of files:

- A collection of source code files containing your class definitions.
- The classes in the standard .NET Framework Class Library are pre-compiled into several libraries
- The most frequently used library is mscorlib.dll stored under C:\Windows\Microsoft.NET\Framework.

Your own source code files need to be compiled into .NET assemblies.

- *compilation unit*.

Namespaces

C# programs and libraries are a collection of classes

To avoid name clashes between these classes, namespaces are used to group relevant classes together

```
namespace NameSpaceName {
  class definitions
}
```

Namespaces can be nested.

"using" directive to include class names from another namespace

A class has a short name and qualified name. For example, in the class Client (see later part of this lecture notes), the short name is "Client" and the qualified name is "ClientDB.Client".

The Global Namespace

In fact, every class is in a namespace.

If a class is not defined inside an explicit namespace, it belongs to the global *nameless* namespace.

The Global Namespace Example

```
class TestApp
{
    // Define a new class called 'System' to cause problems.
    public class System { }

    // Define a constant called 'Console' to cause more problems.
    const int Console = 7;
    const int number = 66;

    static void Main()
    {
        // The following line causes an error. It accesses TestApp.Console,
        // which is a constant.
        //Console.WriteLine(number);
    }
}

// The following line causes an error. It accesses TestApp.System,
// which does not have a Console.WriteLine method.
System.Console.WriteLine(number);

// OK
global::System.Console.WriteLine(number);
```

Compilation Unit

Most source code files contain one or more full class definitions, such as Student.cs.

source code file - a compilation unit,

Some source code files contain partial class definitions.

Classes in a compilation unit - compiled into the same assembly

Information Hiding

One important principle in class design is *information hiding.*

For example, how we represent the name of a client is an internal implementation issue

use three string variables to represent the name (first, second and last names).

fields declared to be private - not accessible (or visible) outside of the class.

- representing a name with three strings is far from ideal - there are better ways of doing it. However if we change the implementation later the other classes that use the client class would not be affected.

Access Specifiers

Control the visibility and accessibility of class members using access specifiers.

Four access specifiers:

- *public*: unlimited access
- *private*: access limited to the containing class
- *protected*: access limited to the containing class or types derived from the containing class
- *internal*: access limited to this program

The default access specifier is private.

Controlled Access

To get the name of the client so the name can be displayed.

- such accessor is called “**get accessor**”

- or to change the name if the old name is found to be incorrect.
    - such accessor is called “**set accessor**”

C# introduced a new kind of members for the above purposes. They are called property.

Properties

A property is a member that provides controlled access to an attribute of an object or a class.

Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a client, and so on.

Properties are named members of a class with associated types, and the syntax for accessing a field and a property is the same.

A property does not denote storage location.

FullName Property

Property declaration:

```
public string FullName
 {
          get
          {
              return firstName + " " +secondName + " " + lastName;
          }

          set
          {
              SetNames(value);
          }
 }
```

The **value** keyword is used to define the value being assigned by the set accessor.

We can read and modify the property, for example:

```
Client c = new Client("Joe Blow");
string name = c.FullName;   // cause get accessor to execute.
c.FullName = "John Smith";  // cause set accessor to execute, value="John Smith".
```

Properties typical pattern

- Typical pattern for accessing fields.

```
private int x;
public int GetX();
public void SetX(int newVal);
```

- Elevated into the language:

```
private int count;
public int Count {
      get { return count; }
      set { count = value; }
  }
```

Accessing fields

- Using a property is more like using a public field than calling a function:

```
FooClass foo;
int count = foo.Count; // calls get
int count = foo.count; // compile error
```

- The compiler automatically generates the routine or in-lines the code.

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get
        {
            return seconds / 3600;
        }
        set
        {
            seconds = value * 3600;
        }
    }
}
```

https://msdn.microsoft.com/en-AU/library/x9fsa0sw.aspx

```
class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Assigning the Hours property
        // causes the 'set' accessor to be called.
        t.Hours = 24;

        // Evaluating the Hours property
        // causes the 'get' accessor to be called.
        System.Console.WriteLine("Time in hours: "
             + t.Hours);
    }
}


// Output: Time in hours: 24
```

Automatic Properties

- C# 3.0 added a shortcut version for the common case (or rapid prototyping) where my get and set just read and wrote to a backing store data element.
- Avoids having to declare the backing store. The compiler generates it for you implicitly.

```
public decimal CurrentPrice { get; set; }
```

Automatic Properties - Example

```
public class Student
{
    private string name;

    public string Name
    {
        get { return name;  }
        set { name = value;  }
    }
}
```

```
public class Student
{
    public string Name { get; set; }
}
```

Types

- Properties can be used in *interfaces*
- Can have three types of a property

  read-write, read-only, write-only

```
// read-only property declaration
//    in an interface.
int ID { get; };
```

Polymorphism

A class may contain several methods with the *same* name, as long as their method signatures are *different*.

For example you may define a math class with the commonly used math functions:

```
public int Average(int x, int y);
public int Average(int x, int y, int z);
public double Average (double x, double y);
public float Average (float x, float y, float z);
```

Example: ClientDB

Assume that we are to design an application to handle a client database (DB).

perform the following operations repeatedly:

- add a new client to DB,
- remove an existing client from DB,
- check how many clients are in DB,
- find out how many clients are living in a given post-code area,
- get the details of an existing client from DB.

one DB object holds (or links to) multiple client objects

Client Objects

A client object would contain:

- the name of the client
- the phone number of the client
- the home address
- the credit information

We want to be able to do such things as

- get or change the name of the client
- get or change the phone number
- modify the client credit information
- check whether two clients are in fact the same person

ClientDB Object

- Links to all clients currently in our DB
- the total number of clients in the DB
- the capacity of the DB

we would like to be able to:

- add (or remove) a client to (or from) the DB
- tell us the total number of clients
- tell us whether a given person is one of the clients in the DB
- find out how many clients live in the given post-code area

Modelling the Clients

One class to model any client.

Each client will only have a name (including first name, last name etc).

You may add phone number, home address etc

Each client can get and change its name, can confirm whether a given name is the same as its own name.

Create a ClientDB Project

Start Visual Studio.

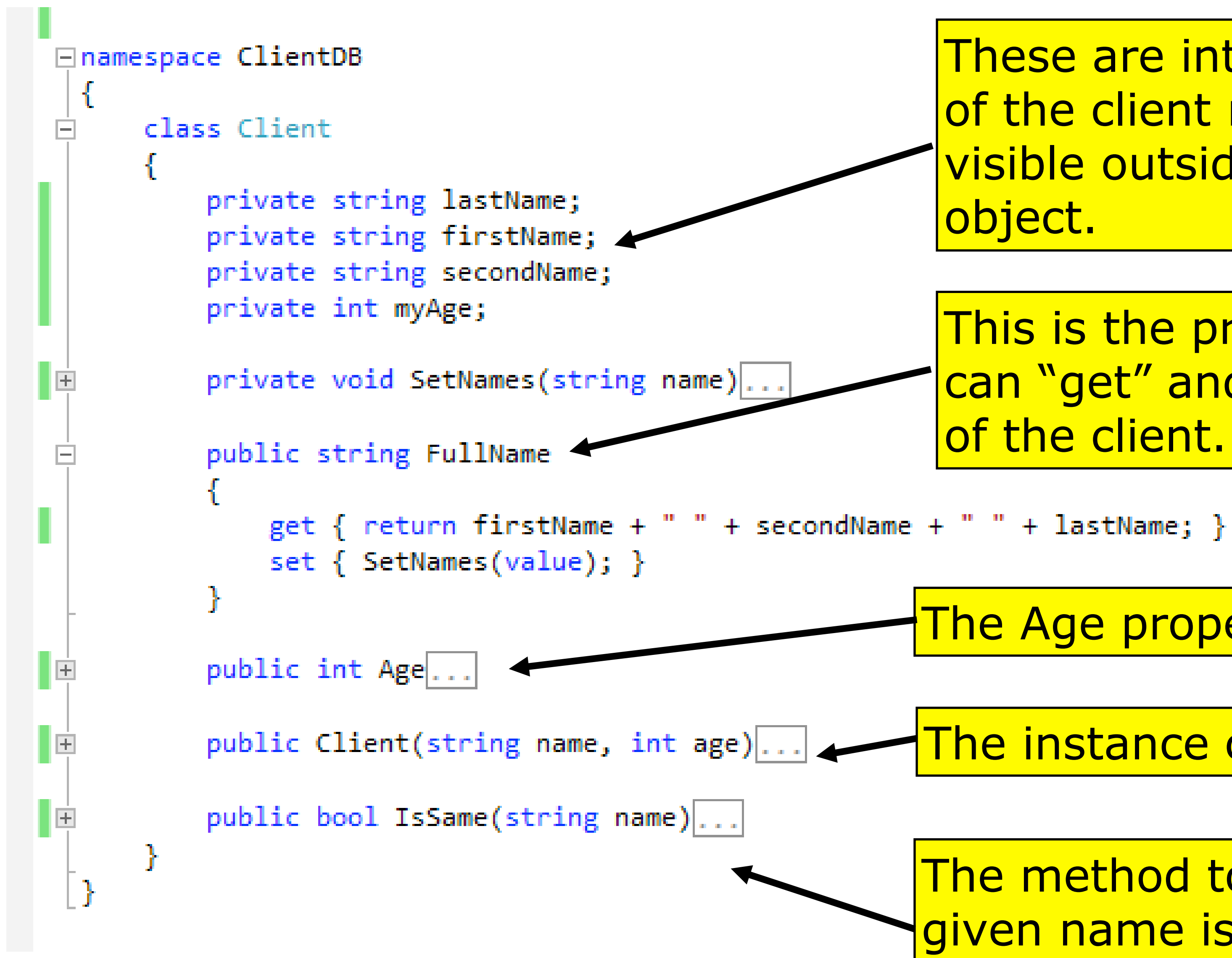
Create a new project - select Windows Forms Application template

change the project name from the default “WindowsFormsApplication1” to “ClientDB”

Bring up “Solution Explorer” (View =>Solution Explorer)

Add a new class to the project by right-clicking ClientDB project from Solution Explorer, then select Add=>class, then select the “Class” template, change the name of the class from “Class1.cs” to “Client.cs”.

Client class



```
namespace ClientDB
{
    class Client
    {
        private string lastName;
        private string firstName;
        private string secondName;
        private int myAge;

        private void SetNames(string name)...

        public string FullName
        {
            get { return firstName + " " + secondName + " " + lastName; }
            set { SetNames(value); }
        }

        public int Age...

        public Client(string name, int age)...

        public bool IsSame(string name)...
    }
}
```

These are internal representation of the client name. They are not visible outside of the class or object.

This is the property - the public can “get” and “set” the FullName of the client.

The Age property

The instance constructor

The method to check whether the given name is same as this client’s name

Splitting a String

We need to break the string such as “John Smith” into “John” and “Smith” to separate the first name from the last name.

Standard string handling method “split” can be used to separate a string based on the string separators,

space “ ” and dot “.”

Assume that the string object `name` contains the string “John Smith”, the following statement will create a new array of strings `names` that contains the string “John” and string “Smith”:

```
string[] names = name.Split( new string[]{" ", "."},
                 StringSplitOptions.RemoveEmptyEntries );
```

Separating Names

```
private void SetNames(string name)
{
     string[] names = name.Split(new string[]{" ", "."},
          StringSplitOptions.RemoveEmptyEntries);
     int n = names.Length;
     // Length is the number of array elements (names)

     if (n==1) { firstName = names[0]; }
     else if (n==2) {
        firstName = names[0]; lastName = names[1];
     }
     else {
       firstName = names[0]; secondName = names[1];
       lastName = names[n-1];
      }
}
```

Note: careful readers will notice an error here: what happens if the string name is empty? How would you correct this error?

Same Name?

```
public bool IsSame(string name)
 {
    string[] names = name.Split(new string[]{" ", "."},
            StringSplitOptions.RemoveEmptyEntries);
    int n = names.Length;
    return lastName.Equals(names[n-1]) &&
        firstName.Equals(names[0]);
  }
```

Again there is a serious bug in the above code

ClientDB class

private constant and fields:

```
    private string companyName;
    private int totalClients;  // total number of clients maintained by this object
    private const int MAX_CLIENTS = 100;  // constant
    private Client[] myClients;  // an array containing the clients' object references
```

Instance constructor

```
    public ClientDB(string companyName)
    {
         this.companyName = companyName;
         totalClients = 0;
         myClients = new Client[MAX_CLIENTS];
     }
```

Note in the above, the reserve word "this" represents the current object (the object reference of the current object).

ClientDB - AddClient

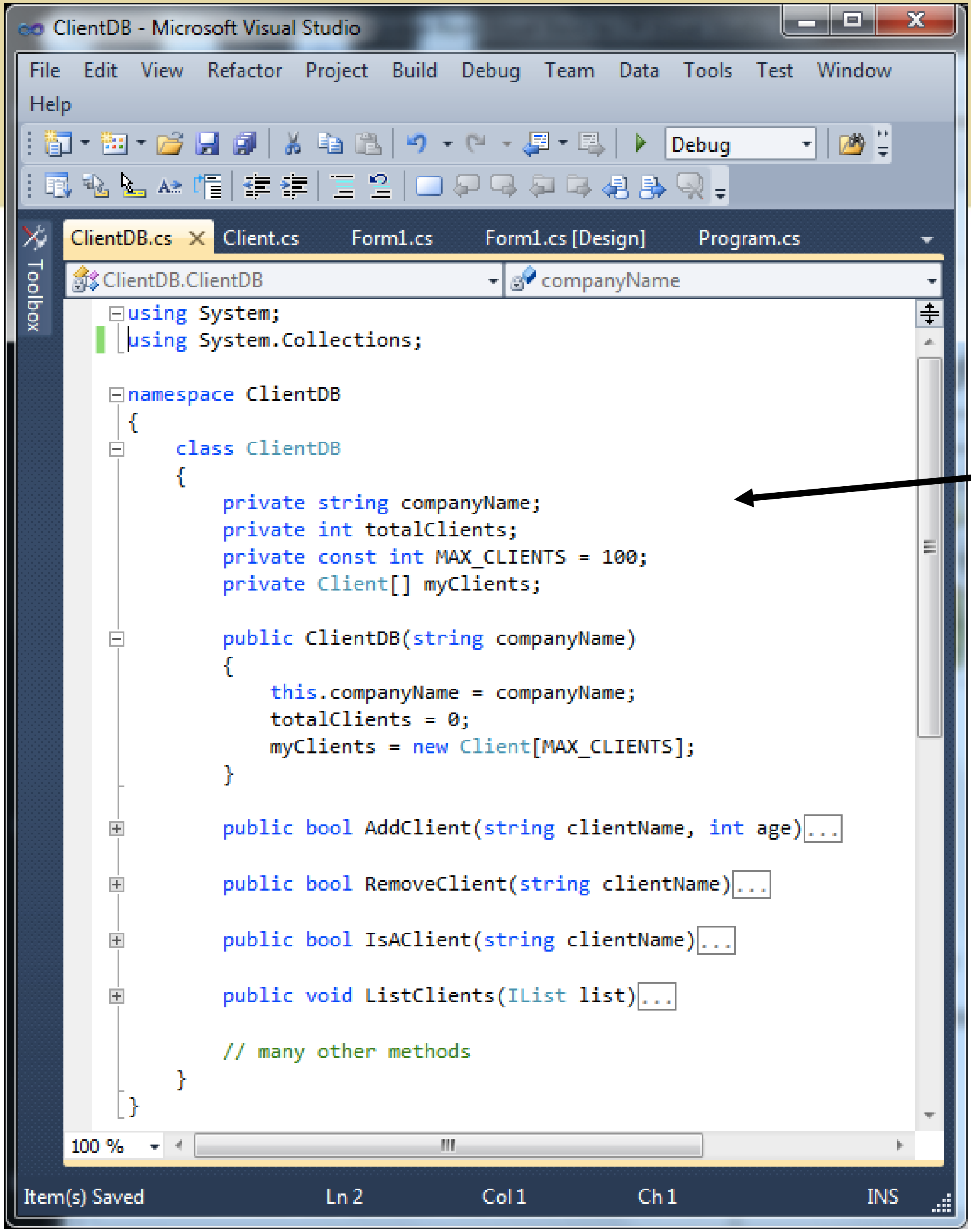
```
public bool AddClient(string clientName)
 {
       // find any empty slot in myClients array
       int index = -1;
       for (int i=0; i<MAX_CLIENTS; ++i)
                if (myClients[i] == null)  {
                    index = i; break;
                }
       if (index != -1) {
                myClients[index] = new Client(clientName);
             // create a new client object

             ++totalClients;
             return true;
       }
       return false;
}
```

ClientDB - RemoveClient

```
public bool RemoveClient(string clientName)
 {
       for (int i = 0; i < MAX_CLIENTS; ++i)  {
             if (myClients[i] != null) {
                    if (myClients[i].IsSame(clientName)) {
                           myClients[i] = null;
           // this client object is to be garbage collected
                           --totalClients;
                           return true;
                    }
               }
         }
         return false;
  }
```

ClientDB - IsAClient

```
public bool IsAClient(string clientName)
{
        for (int i = 0; i < MAX_CLIENTS; ++i) {
              if (myClients[i] != null) {
                    if (myClients[i].IsSame(clientName)) {
                        return true;
                    }
               }
          }
          return false;
 }
```

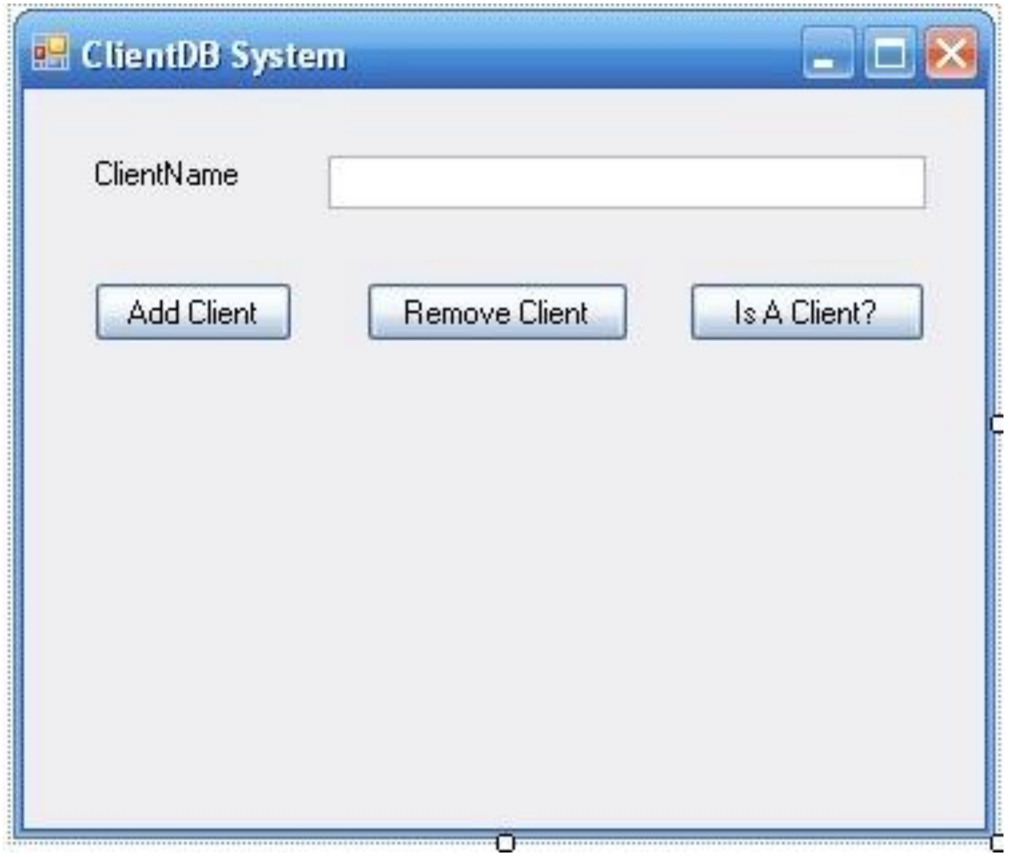


```
using System;
using System.Collections;

namespace ClientDB
{
    class ClientDB
    {
        private string companyName;
        private int totalClients;
        private const int MAX_CLIENTS = 100;
        private Client[] myClients;

        public ClientDB(string companyName)
        {
            this.companyName = companyName;
            totalClients = 0;
            myClients = new Client[MAX_CLIENTS];
        }

        public bool AddClient(string clientName, int age)...

        public bool RemoveClient(string clientName)...

        public bool IsAClient(string clientName)...

        public void ListClients(IList list)...

        // many other methods
    }
}
```

Use Solution Explorer to add another class “ClientDB.cs” to the project.

Graphical User Interface

Use the Windows Forms Designer to draw the following window. The corresponding source code file is named Form1.cs.

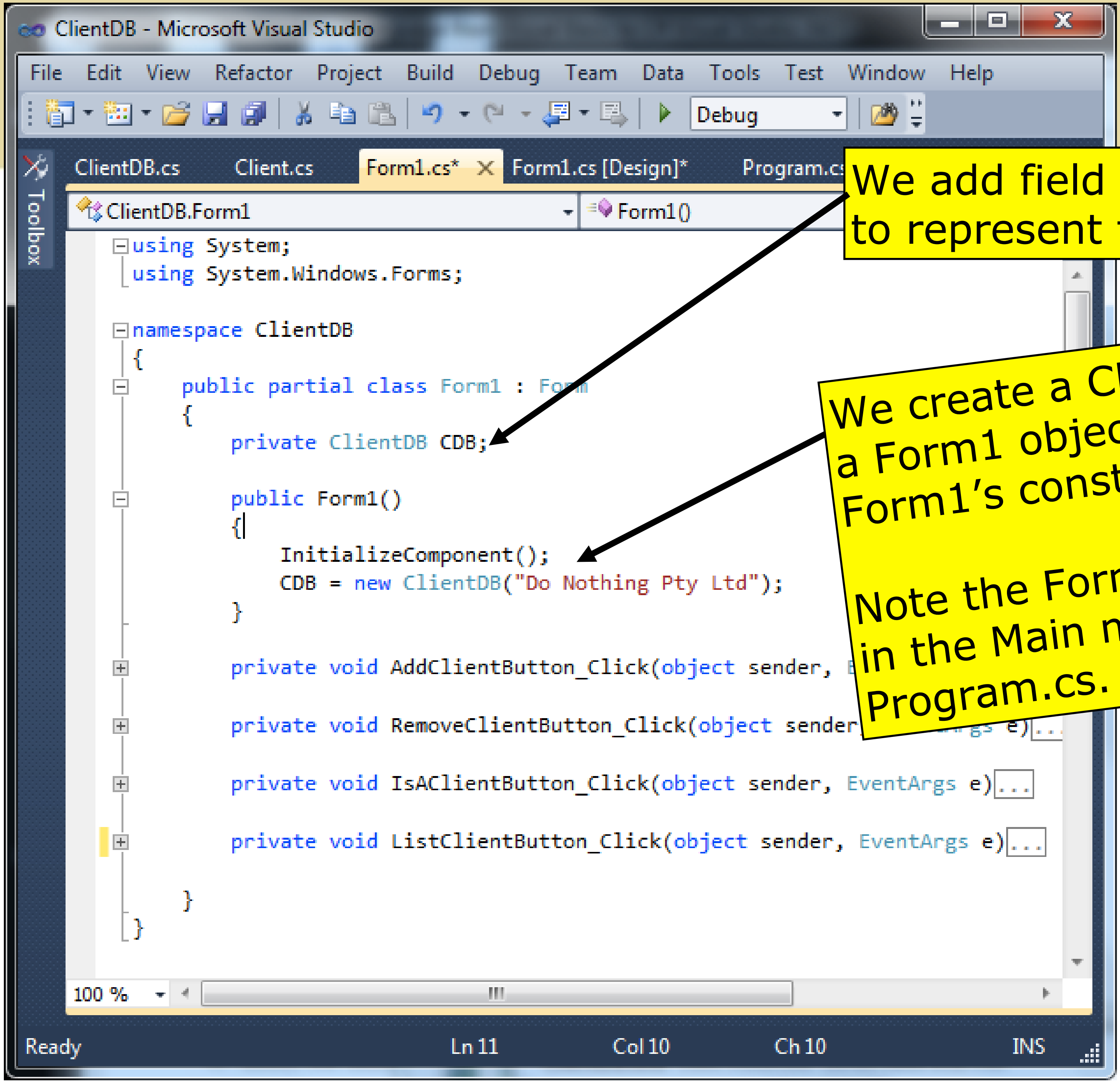


Modify Form1.cs

In Form1.cs class, we add a field to represent the ClientDB object: CDB

When the Form1 is instantiated in Program.cs, its constructor is executed. We add a line (in red colour) in From1's constructor to create a ClientDB object:

```
public Form1(
{
    InitializeComponent();
    CDB = new ClientDB("Do Nothing Pty Ltd");
    }
```



```
using System;
using System.Windows.Forms;

namespace ClientDB
{
    public partial class Form1 : Form
    {
        private ClientDB CDB;

        public Form1()
        {
            InitializeComponent();
            CDB = new ClientDB("Do Nothing Pty Ltd");
        }

        private void AddClientButton_Click(object sender, E

        private void RemoveClientButton_Click(object sender

        private void IsAClientButton_Click(object sender, EventArgs e)...

        private void ListClientButton_Click(object sender, EventArgs e)...

    }
}
```

We add field CDB in Form1 class to represent the client database.

We create a ClientDB object when a Form1 object is created (inside Form1's constructor).

Note the Form1 object is created in the Main method of Program.cs.

Event Handler for AddClientButton

```
  private void AddClientButton_Click(object sender,
EventArgs e)
  {
      if (!CDB.AddClient(ClientNameTextBox.Text))
      {
         MessageBox.Show("Cannot add this client!");
      }
  }
```

Event Handler for RemoveClientButton

```
  private void RemoveClientButton_Click(object
sender, EventArgs e)
  {
      if (!CDB.RemoveClient(ClientNameTextBox.Text))
     {
      MessageBox.Show("Cannot remove this client!");
      }
  }
```

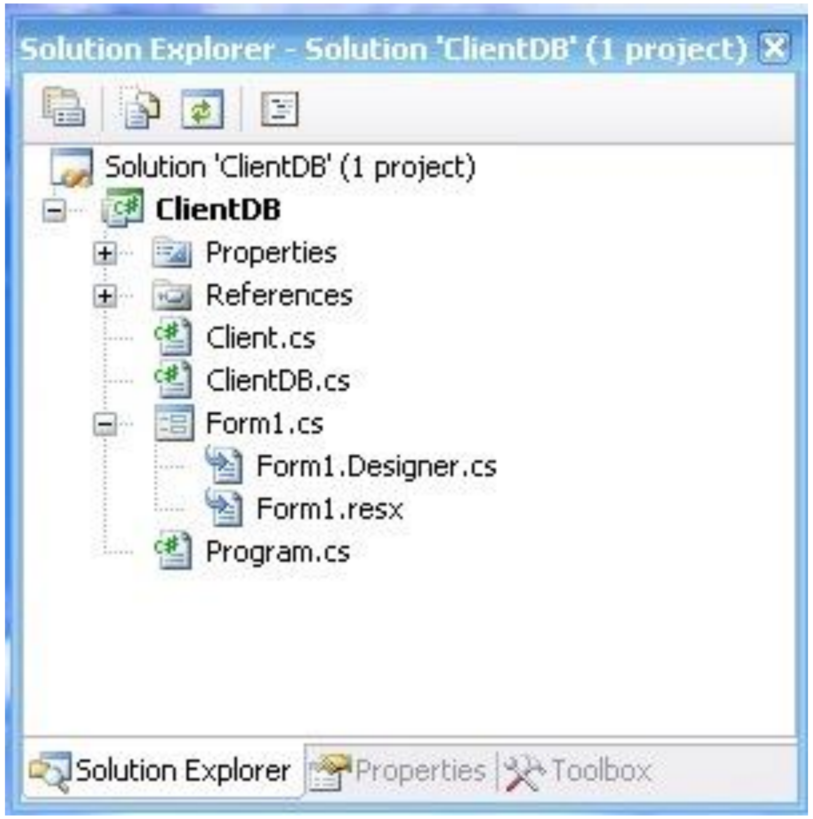
Event Handler for IsAClientButton

```
   private void IsAClientButton_Click(object
sender, EventArgs e)
   {
          if
     (CDB.IsAClient(ClientNameTextBox.Text))
     {
          MessageBox.Show("Our client!");
     }
     else
     {
          MessageBox.Show("Not our client!");
     }
 }
```

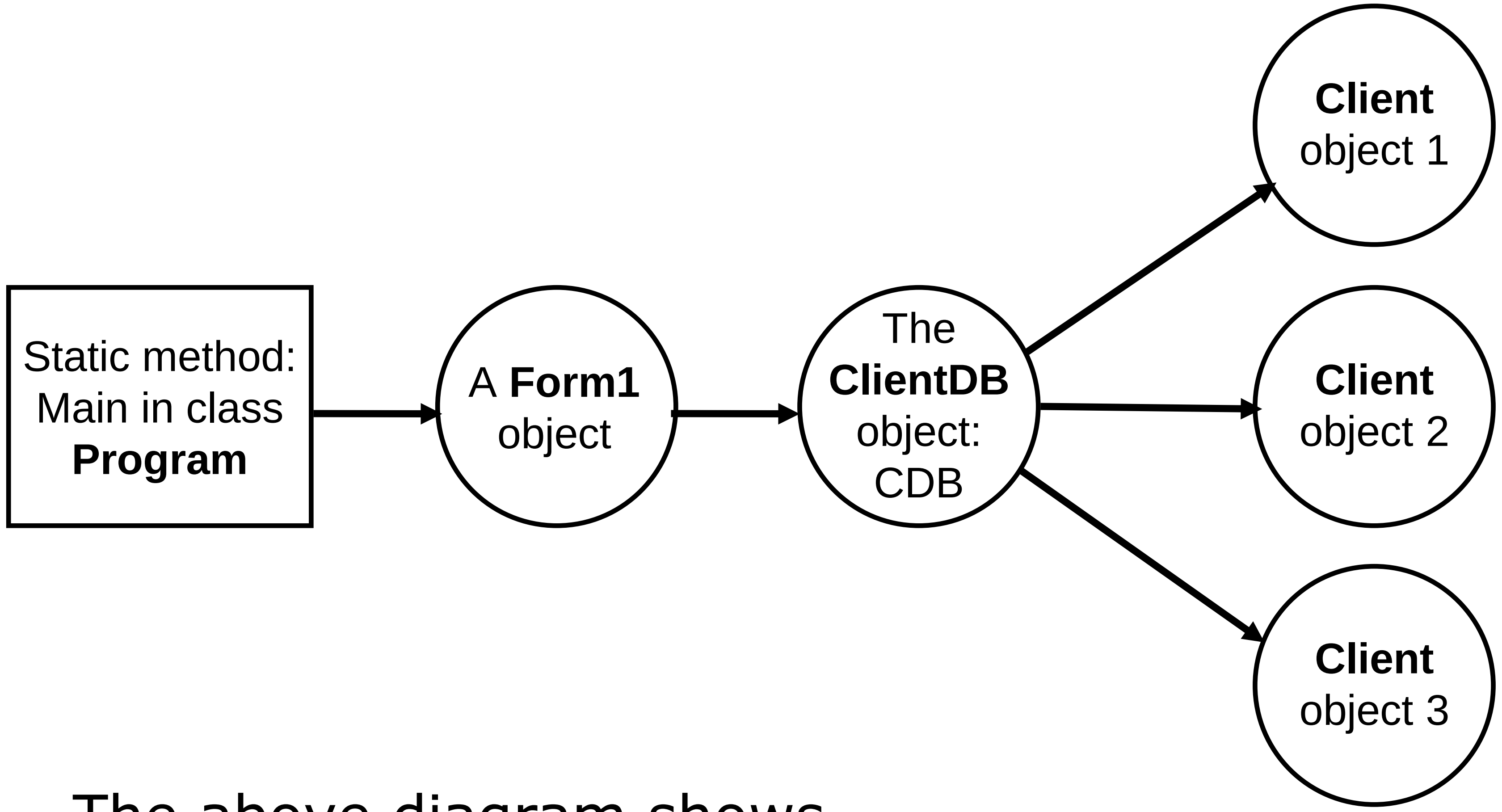
ClientDB Project

The project consists of four source code files:

- Client.cs
- ClientDB.cs
- Form1.cs (in the form of two partial classes)
- Program.cs



Relationship between Objects



The above diagram shows where each object is created.

A Note

The ClientDB example is complex as it consists of several classes spread in different source code files.

Some of the source code in the program is generated by Visual Studio, particularly in Form1.cs and Program.cs.

This is typical of C# programs. You MUST do your best to understand how the whole program comes together and how it works!

Exceptions

An exception is an object created at the run-time to represent an error condition

Exception mechanism provides a more structured, uniform, and general approach to handling error and abnormal conditions.

Exception (2)

Exceptions may be caused by run-time errors such as
number overflow,
array index out of bounds,
incorrect input

The exception mechanism is also used to separate the normal execution flow from exceptional execution flow.

An exception is *thrown* inside a *try block* and is *caught* by a *catch block*

An exception class is derived from the class System.Exception.

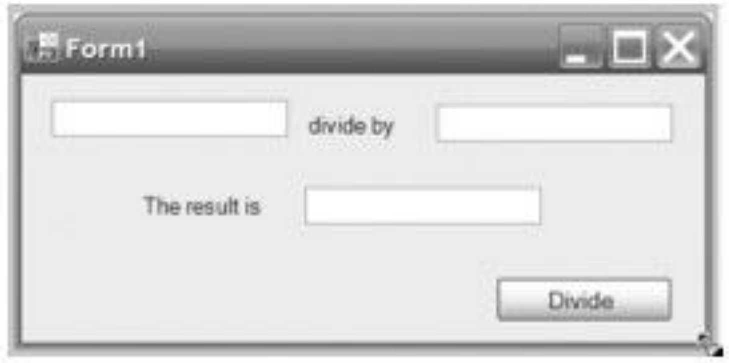
Exception (3)

```
try
{
  // the try block containing normal stuff but the execution stops
  // when an exception is thrown by one of the statements in this block
}
catch ( ExceptionType ex)
{
  // catch block -these statements will be executed if the exception object
  //       thrown from the try block matches the ExceptionType.
  // You can have more than one catch block here
  //       with different exception type.
  // If there is no need to use the exception parameter, you can omit
  //       the parameter list
}
finally
{
  // this finally block is optional. If this block exists, statements
  // here will be executed after the try block executes successfully or
  // after the exception is processed.
}
```

Exception Example (1)

A windows application that does division*:

- the user types in two numbers then clicks button "Divide"
- the program reads the two numbers and performs a division and prints the result in the result box.



Exception Example (2)

```
private void divideButton_Click(object sender,
  EventArgs e)
  {
        double dividend =
    Double.Parse(dividendBox.Text);
        double divisor =
    Double.Parse(divisorBox.Text);
        resultBox.Text = (dividend /
    divisor).ToString();
  }
```

The exception will cause the program to terminate (crash), since we have made no attempt to handle this exception.

Exception Example (3)

```
private void divideButton_Click(object sender, EventArgs e)
{
        try
        {
              double dividend =
                   Double.Parse(dividendBox.Text);
              double divisor =
                  Double.Parse(divisorBox.Text);
              resultBox.Text = (dividend /
                                       divisor).ToString();
        }
       catch
        // we are not using the exception parameter here
       {
             MessageBox.Show("Something went wrong!");
             // but we can't tell you what went wrong!
       }
}
```

A Better Solution

```
private void divideButton_Click(object sender, EventArgs e)
{
        try
        {
              double dividend = Double.Parse(dividendBox.Text);
              double divisor = Double.Parse(divisorBox.Text);
              resultBox.Text = (dividend / divisor).ToString();
        }
        catch (FormatException formatEx)
        {
      MessageBox.Show("You have to enter numbers, not text");
       // this error message is more specific.
        }
        catch (Exception ex)
        {
             MessageBox.Show("Something went wrong : " + ex.Message);
    // although we can't tell what went wrong, we can give an
    // extra message which may help the user to identify the problem.
        }
}
```

finally block

```
private void divideButton_Click(object sender, EventArgs e)
{
          string result = "";
          try
          {
              double dividend = Double.Parse(dividendBox.Text);
              double divisor = Double.Parse(divisorBox.Text);
              result = (dividend / divisor).ToString();
         }
         catch (FormatException formatEx)  {
              MessageBox.Show("You have to enter numbers, not text");
         }
         catch (Exception ex)
         {
              MessageBox.Show("Something went wrong : " + ex.Message);
         }
         finally
         {
            resultBox.Text = result;
    // this block is executed whether there is
    // an exception (in which case the result box is not assigned,
    // so it remains empty). If there is no exception, we assign
    // the result to the result box)
         }
}
```

Throwing an Exception

A user program can also throw an exception:

- define an exception class by subclassing `System.Exception`
- create an exception object using the new exception class, or simply using System.Exception class

```
Exception ex = new Exception("A fatal
 error!");
```

- throw the exception object:

```
throw ex;
```

C# Reference

- Great resource for C#
- https://msdn.microsoft.com/en-us/library/618ayhy6.aspx

- E.g. **C# Programming Guide**
- https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx

- E.g. **Interfaces (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173156.aspx

- **Classes and Structs (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173109.aspx

- **Polymorphism (C# Programming Guide)**
- https://msdn.microsoft.com/en-us/library/ms173152.aspx

- **Abstract and Sealed Classes and Class Members (C# Programming Guide)**
- https://msdn.microsoft.com/en-au/library/ms173150.aspx

- **Properties (C# Programming Guide)**
- https://msdn.microsoft.com/en-AU/library/x9fsa0sw.aspx

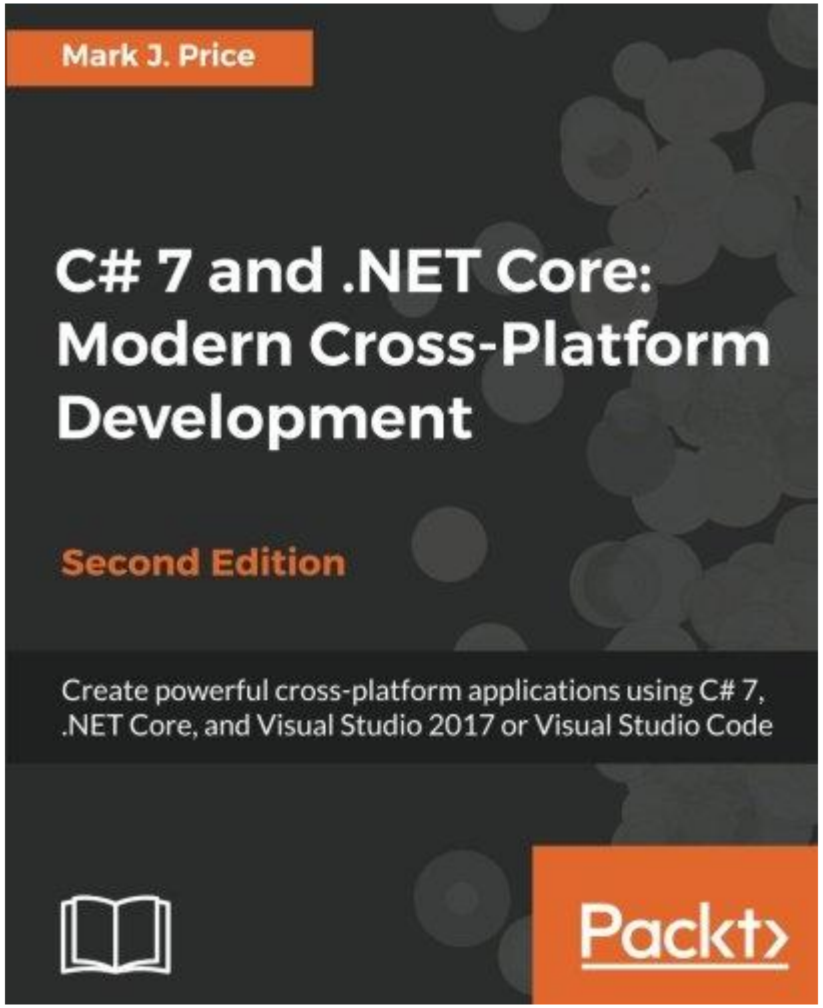
Reading/ reference

You should be okay with Chapters 1-5

Specifically, review:

Chapter 6. Building Your Own Types with Object-Oriented Programming

Chapter 7. Implementing Interfaces and Inheriting Classes


Mark J. Price

C# 7 and .NET Core: Modern Cross-Platform Development

Second Edition

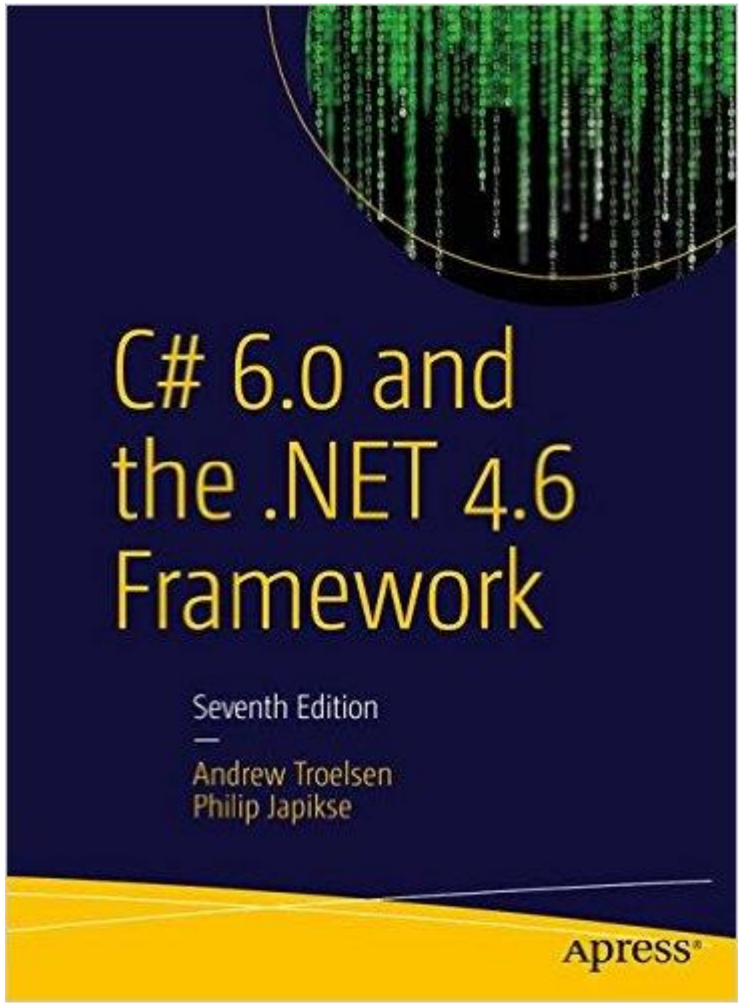
Create powerful cross-platform applications using C# 7, .NET Core, and Visual Studio 2017 or Visual Studio Code

Packt>

Reading/ reference

Nothing this week, but take a look at the book anyway, its really good..

We will be referring to it later.

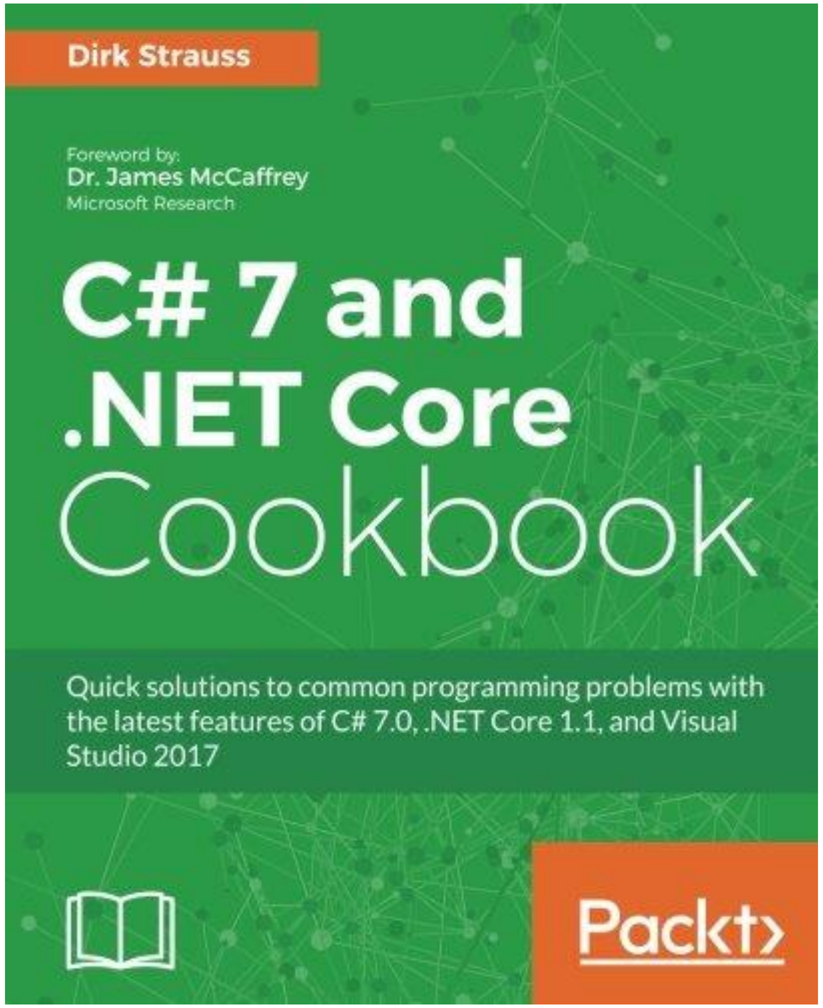


Reading/ reference

You could skim through these:

Chapter: CLASSES AND GENERICS

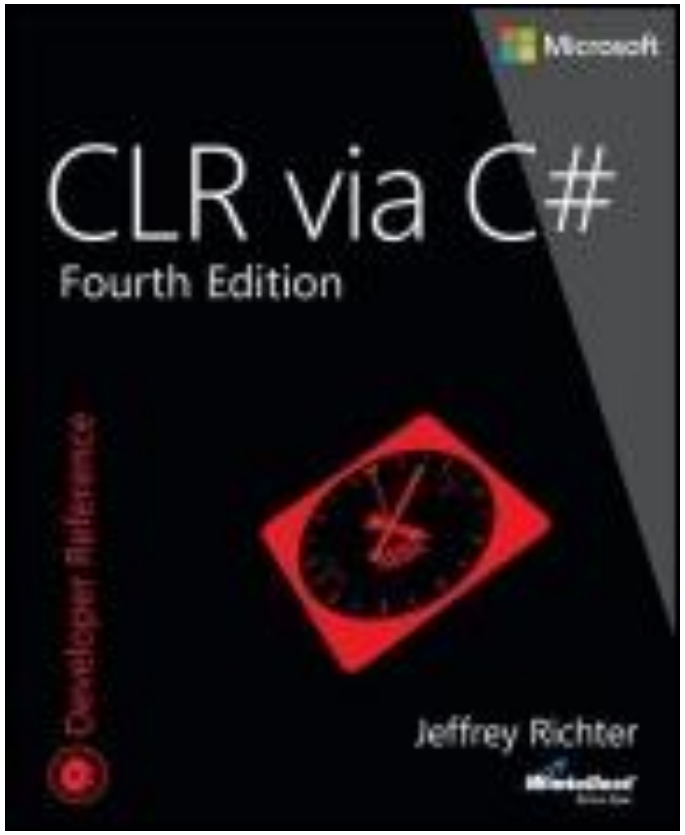
Chapter: OBJECT-ORIENTED PROGRAMMING IN C#


Dirk Strauss
Foreword by:
Dr. James McCaffrey
Microsoft Research
C# 7 and .NET Core Cookbook
Quick solutions to common programming problems with the latest features of C# 7.0, .NET Core 1.1, and Visual Studio 2017
Packt>

Reading/ reference

We will be covering Interfaces more later on, but if you want to look at this:

Chapter 13. Interfaces



Widening and Narrowing

Murdoch UNIVERSITY

Few more concepts

- Assigning an object to an ancestor reference is considered to be a ***widening*** conversion, and can be performed by simple assignment

```
Holiday day = new SummerHoliday();
```

- Assigning an ancestor object to a reference can also be done, but it is considered to be a ***narrowing*** conversion and must be done with a cast:

```
SummerHoliday summerDay= new SummerHoliday();
Holiday day = summerDay;
SummerHoliday summerDay = (SummerHoliday)day;
```

Widening and Narrowing

- Widening conversions are most common.

  Used in polymorphism.
- Note: Do not be confused with the term widening or narrowing and memory. Many books use *short* to *long* as a widening conversion. A *long* just happens to take-up more memory in this case.
- More accurately, think in terms of sets:

  The set of animals is greater than the set of parrots.

  The set of whole numbers between 0-65535 (ushort) is greater (wider) than those from 0-255 (byte).

Type Unification

- **Everything** in C# inherits from `object`

  Similar to Java except includes value types.

  Value types are still light-weight and handled specially by the CLI/CLR.

  This provides a single base type for all instances of all types.

  Called **Type Unification**