

## Diagrams and Screen snapshots from the Course.

This document contains screen snapshots I took during the course. There is a mixture of annotated code and diagrams. At the top of each page, I indicate the module in the course a diagram applies to, and I usually write something about it.

What you're looking at is a PDF document. If you want a better browsing experience you can download it, zoom in, whatever.

Mark.

## MODULE 1.

When does this loop end?

```
for(int loopCounter = 0; loopCounter < 10; loopCounter++)
{
    Console.Out.WriteLine("loopCounter is at: " + loopCounter);
    total = total + loopCounter;
}
Console.Out.WriteLine("Total of values is: " + total);
Console.Out.WriteLine("Average is: " + total / 10);
```

console.Out.WriteLine(\$"Total of values is: {total}");

## MODULE 1.

What is going on here?

It is not a class

it contains related data and functionality

what is that? Constructor.

```
C# Example
public struct name
{
    string firstName;
    string middleName;
    string lastName;
    string suffix;
    public name(string first, string middle, string last, string suff)
    {
        firstName = first;
        middleName = middle;
        lastName = last;
        suffix = suff;
    }
    public string getFullName()
    {
        return firstName + " " + middleName + " " + lastName + " " + suffix;
    }
}
```

name Person1 = new name("Zoe", "ella","wong","Miss");

## MODULE 1.

C# Example

```
public class Person
{
    public float _height;
    public float _weight;
    public string _ethnicity;
    public string _gender;
    public Person()
    {
        _height = 5.7F;
        _weight = 198.6F;
        _ethnicity = "Doesnt matter";
        _gender = "male";
    }
    public Person(float height, float weight, string ethnicity, string gender)
    {
        _height = height;
        _weight = weight;
        _ethnicity = ethnicity;
        _gender = gender;
    }
    public void Walk()
    {
    }
    public void Run()
    {
    }
    public void Eat()
    {
    }
    public void Sit()
    {
    }
    public void Speak()
    {
    }
}
```

"Blueprint for creating objects"

What is going on here?

Person p = new Person();

Person p2 = new person(180,45,"AI","NB");

method (function)

what are the differences between a struct and a class?  
- structs are value types (They are stored in stack memory)  
- classes are reference types (They are stored in heap memory)  
- Capabilities to do with inheritance.

## MODULE 1.

```
C# Example
public class Person
{
    public float _height;
    public float _weight;
    public string _ethnicity;
    public string gender;
    public Person()
    {
        _height = 5.7F;
        _weight = 198.6F;
        _ethnicity = "Doesnt matter";
        _gender = "male";
    }
    public Person(float height, float weight, string ethnicity, string gender)
    {
        _height = height;
        _weight = weight;
        _ethnicity = ethnicity;
        _gender = gender;
    }
    public void Walk()
    {
    }
    public void Run()
    {
    }
    public void Eat()
    {
    }
    public void Sit()
    {
    }
    public void Speak()
    {
    }
}
```

What is going on here?

Person p = new Person();

Person p2 = new person(180,45,"AI","NB");

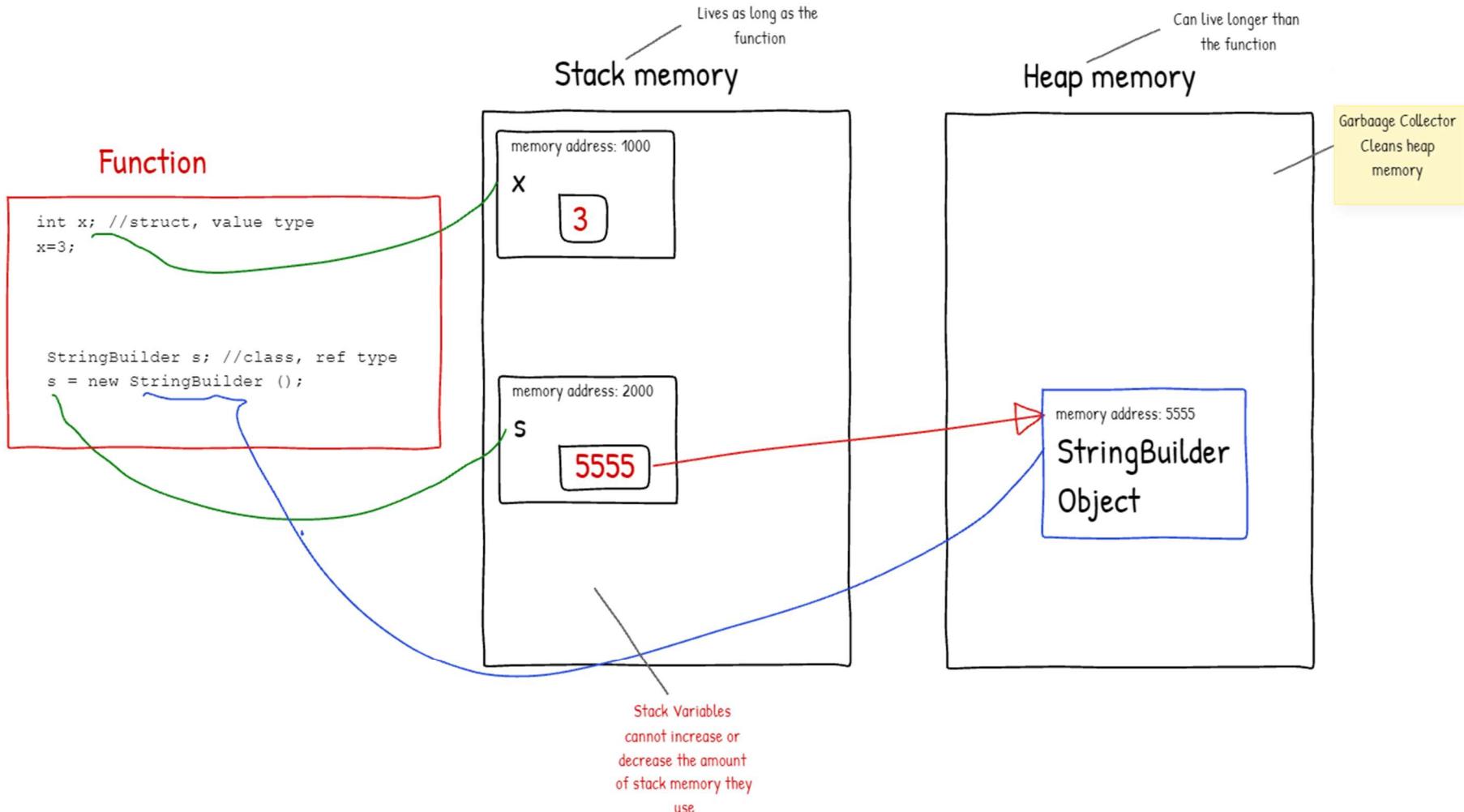
method (function)

what are the differences between a struct and a class?

- structs are ~~value~~ types (They are stored in stack memory)
- classes are reference types (They are stored in heap memory)
- Capabilities to do with inheritance.|

## MODULE 1.

### Value Type Vs Reference Type



## MODULE 1.

What is *Polymorphism*?



When 2 or more objects have the same interface they can be used interchangeably.]

## MODULE 1.

### PolyMorphism in everyday life



What is going on here?

```
C# Example  
abstract class Person  
{  
    // public methods  
    public virtual void eat()  
    {  
        Console.WriteLine("slurping");  
    }  
    public void sleep()  
    {  
        Console.WriteLine("Snoring");  
    }  
    abstract public void move();  
}  
class Student : Person  
{  
    public override void move()  
    {  
        Console.WriteLine("Walking");  
    }  
    public override void eat()  
    {  
        Console.WriteLine("Chewing");  
    }  
}
```

Derived classes can override this

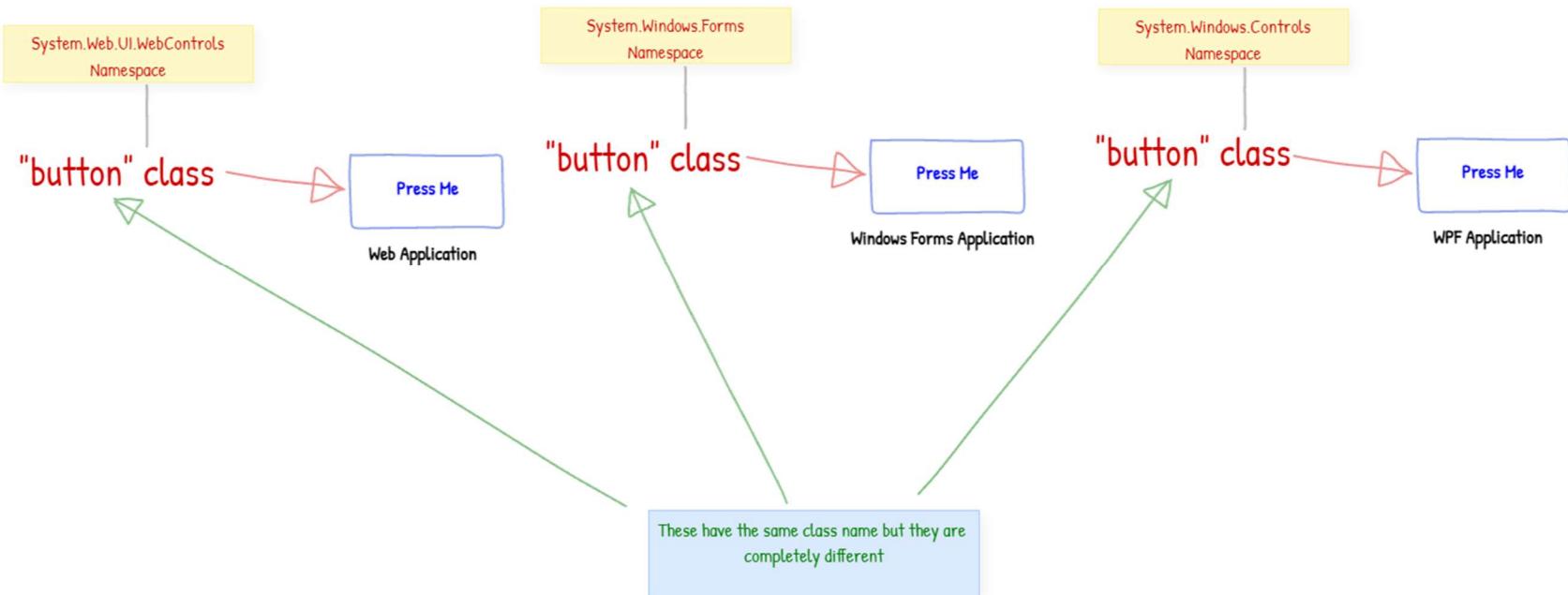
Derived classes must override this

Student is a kind of Person

Student S = new Student();  
S.eat();

## MODULE 1.

"Namespaces" in the .net framework organise datatypes (classes, structures, etc) so that they are easy for us to find and they have a context. In this diagram you see there are multiple classes called "button". Namespaces distinguish between each of them.



# Creating and Using Arrays

page 1-23,24

string[] args)

- C# supports:
  - Single-dimensional arrays
  - Multidimensional arrays
  - Jagged arrays

Tell me anything you can remember about Arrays:

- Every element in the array has the same data type.
- Arrays cannot increase or decrease the number of elements.
- Arrays are objects, they are created in Heap memory.
- An array is a collection of elements that are logically related.
- Arrays are contiguous storage, therefore they are fast!

- Creating an array

```
int[] arrayName = new int[10];
```

- Accessing data in an array:

- By index

```
int result = arrayName[2];
```

subscript (a.k.a. indexer)

- In a loop

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

Activate Win  
Go to Settings tc

## MODULE 1.

### The 'Ternary' operator in C#. (? :)

```
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"The largest integer is {int.MaxValue}");
            Console.WriteLine($"The largest long is {long.MaxValue}");
            Console.WriteLine($"The largest float is {float.MaxValue}");
            Console.WriteLine($"The largest double is {double.MaxValue}");
            Console.WriteLine($"The largest decimal is {decimal.MaxValue}");

            Console.WriteLine(DateTime.Now.Hour>12?"It's afternoon":"It's Morning");
            Console.ReadLine();
        }
    }
}
```

return this if true

evaluate

return this if false|

The diagram illustrates the flow of the ternary operator. A red line starts at the opening parenthesis of the expression `DateTime.Now.Hour>12?`, goes up to the colon, then down to the closing parenthesis, and finally right to the string "It's afternoon". Another red line starts at the opening parenthesis of the expression `DateTime.Now.Hour>12?`, goes up to the colon, then down to the closing parenthesis, and finally right to the string "It's Morning". A third red line starts at the opening parenthesis of the expression `DateTime.Now.Hour>12?`, goes up to the colon, then down to the closing parenthesis, and finally right to the vertical bar at the end of the line. The word "evaluate" is written below the first red line, indicating the point where the condition is checked.

## MODULE 1.

### Passing arguments when starting an application

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays the code for a C# console application named 'ConsoleApp1'. The code in 'Program.cs' is as follows:

```
1
2
3     using System;
4
5     namespace ConsoleApp1
6     {
7         class Program
8         {
9             static void Main(string[] args)
10            {
11                Console.WriteLine($"Hello {args[0]}");
12                Console.ReadKey();
13            }
14        }
15    }
```

The 'Output' window at the bottom left shows the command used to run the application: "C:\Users\Admin>C:\junk\ConsoleApp1\ConsoleApp1\bin\Debug\ConsoleApp1.exe "Zoe"".

The 'Command Prompt' window at the bottom right shows the application's response: "Hello Zoe". The word "Zoe" is highlighted with a red box.

## MODULE 1.

Be careful with the datatypes you choose.

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, Help, and Full Screen. The title bar shows "Program.cs" and "ConsoleApp2.Program". The code editor displays the following C# code:

```
7  namespace ConsoleApp2
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.WriteLine($"The largest integer is {int.MaxValue}");
14             Console.WriteLine($"The largest long is {long.MaxValue}");
15             Console.WriteLine($"The largest double is {double.MaxValue}");
16             Console.WriteLine($"The largest decimal is {decimal.MaxValue}");
17
18             Console.ReadLine();
19         }
20     }
21 }
22
```

A red annotation with a wavy line points from the text "Do not use this for financial." to the line "Console.WriteLine(\$"The largest double is {double.MaxValue}");". Another red annotation with a wavy line points from the text "Use this for financial" to the line "Console.WriteLine(\$"The largest decimal is {decimal.MaxValue}");".

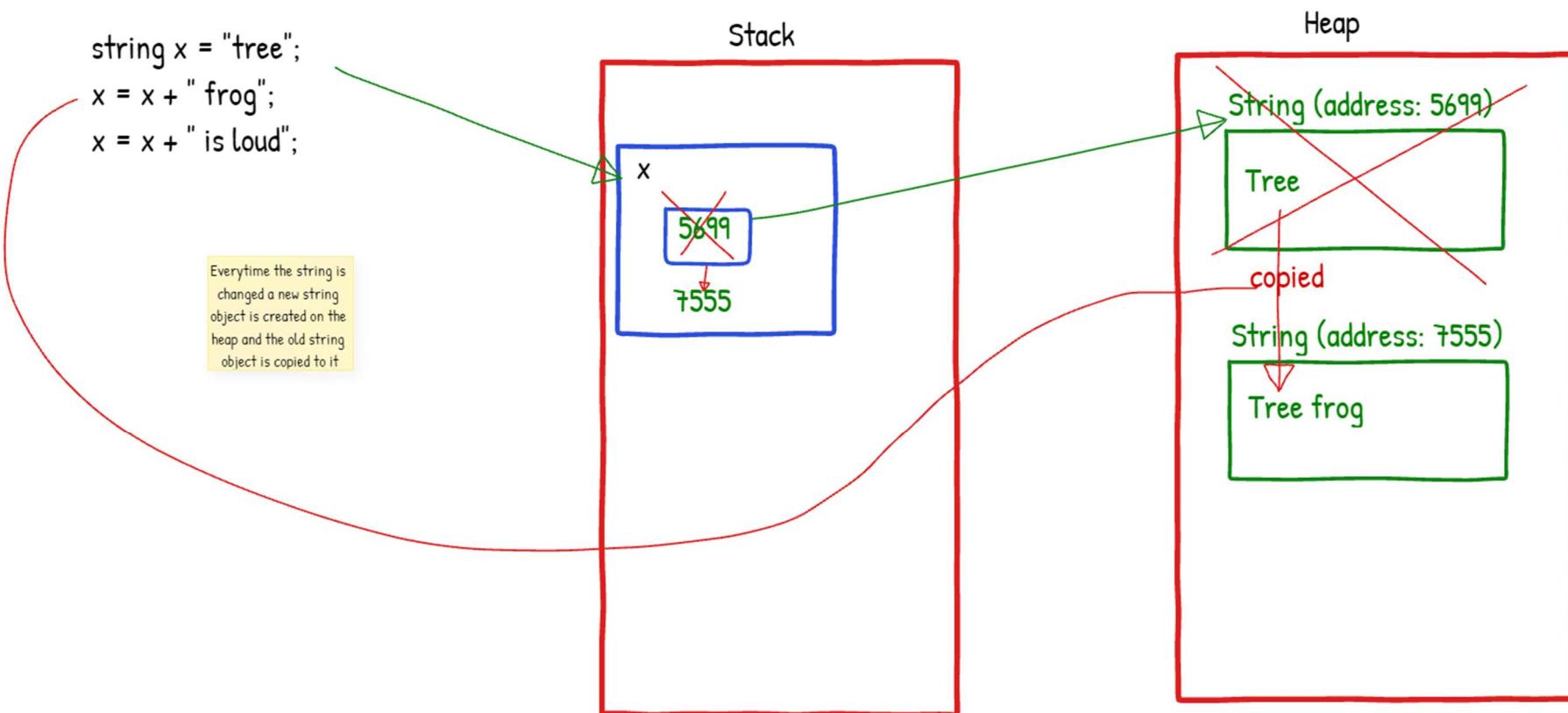
The output window below shows the execution results:

```
C:\junk\ConsoleApp2\ConsoleApp2\bin\Debug\ConsoleApp2.exe
The largest integer is 2147483647
The largest long is 9223372036854775807
The largest double is 1.79769313486232E+308
The largest decimal is 79228162514264337593543950335
```

The status bar at the bottom left indicates "121 %". The Output window shows the message "Skipped loading symbols. Module is".

## MODULE 1.

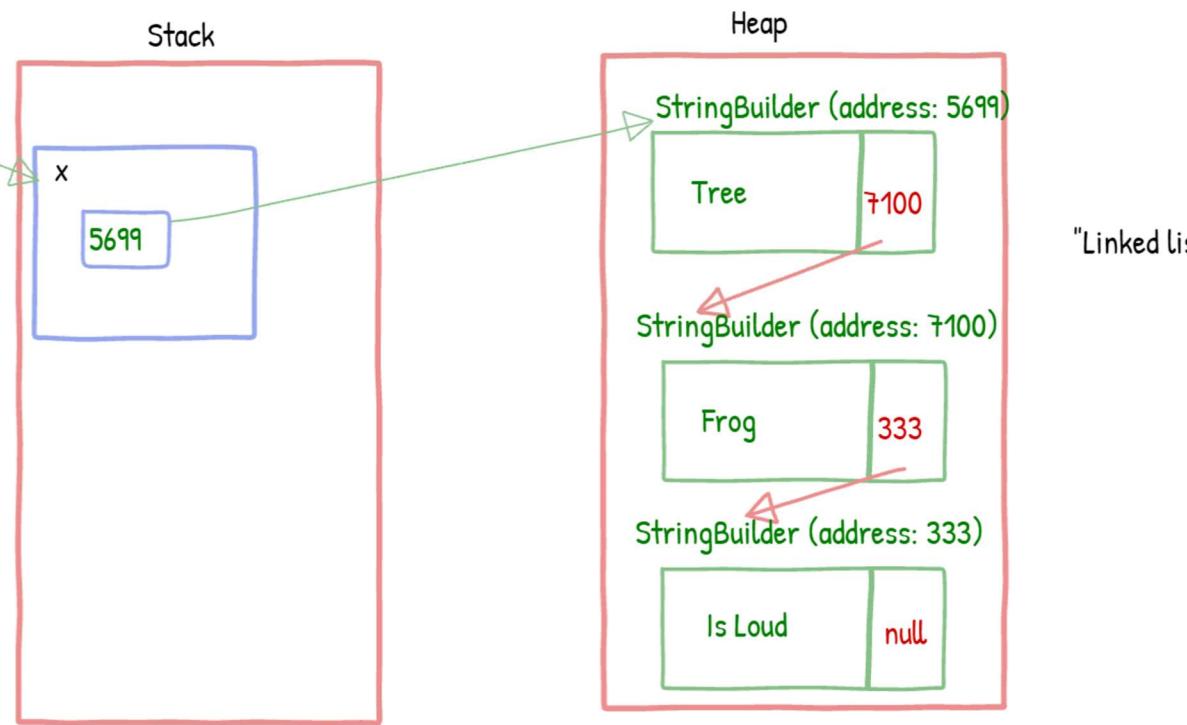
Many changes to a string can result in slowing down your application and the device it is running on. In this case whenever a string is changed in code a new string object must be created and the old string contents copied to it; why? Because strings are arrays, and arrays cannot change in size. All this copying palaver slows things down.



## MODULE 1.

If you are making many changes to a string then use the string builder class. See how it adds a new object each time for the next part of the string and a pointer is updated to indicate where the new object is?

```
StringBuilder x = new StringBuilder();
x.add("Tree");
x.add(" frog");
x.add(" is loud");
```



"Linked list"

# Creating Methods

page 2-3,4

- Methods comprise two elements:
  - Method specification (return type, name, parameters)
  - Method body
- Use the **ref** keyword to pass parameter references

```
void StartService(int upTime, bool shutdownAutomatically)
{
    // Perform some processing here.
}
```

## MODULE 2.

“named parameters”. In this case I wanted to pass a value for `d` but not for `c`

The diagram illustrates the argument mapping for the call to `AddThem(44, 55, d:6)`. The arguments are mapped as follows:

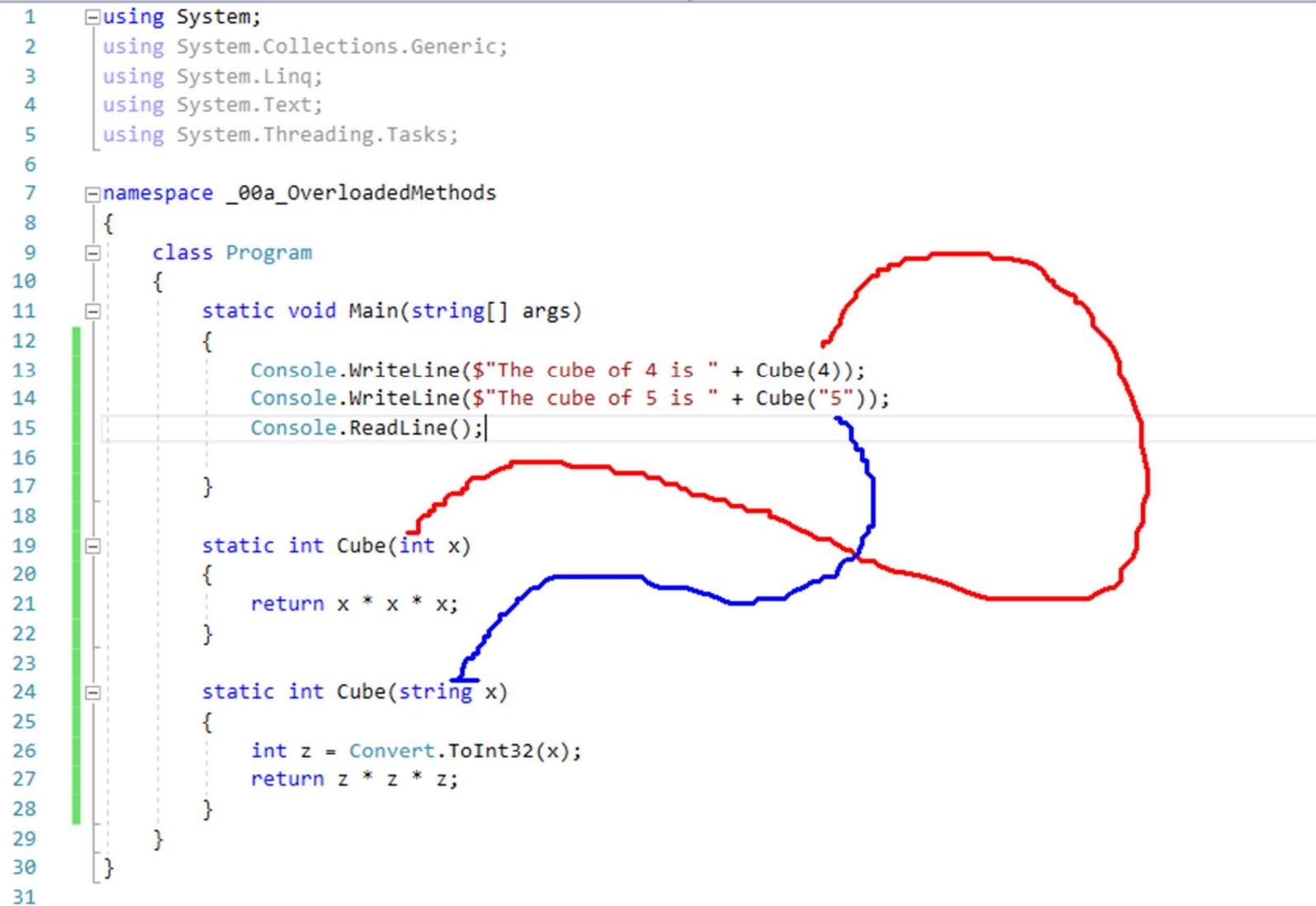
- `44` maps to `a`
- `55` maps to `b`
- `d:6` maps to `c=6` (since `d=0` is the default value)
- `Console.ReadLine()` is not mapped to any parameter in `AddThem`.

```
Program.cs* 00b_OptionalParameters _00b_OptionalParameters.Program Main(string[] args)
7  namespace _00b_OptionalParameters
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.WriteLine(AddThem(44, 55, 77, 88));
14             Console.WriteLine(AddThem(44, 55, 77));
15             Console.WriteLine(AddThem(44, 55));
16             Console.WriteLine(AddThem(44, 55, d:6)); // Argument d:6 is highlighted with a red box
17
18             Console.ReadLine();
19
20
21         }
22
23         static int AddThem(int a, int b, int c=0, int d=0)
24         {
25             return a + b + c + d;
26         }
27     }
28
29 }
```

## MODULE 2.

When two methods have the same name but a different number of parameters and/or different parameter types they are said to be **overloaded**. Can you work out why we have this concept?

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace _00a_OverloadedMethods
8  {
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             Console.WriteLine($"The cube of 4 is " + Cube(4));
14             Console.WriteLine($"The cube of 5 is " + Cube("5"));
15             Console.ReadLine();
16         }
17
18         static int Cube(int x)
19         {
20             return x * x * x;
21         }
22
23         static int Cube(string x)
24         {
25             int z = Convert.ToInt32(x);
26             return z * z * z;
27         }
28
29     }
30 }
31 }
```



## MODULE 2.

### An example of an OUT parameter

```
{  
    public MainWindow()  
    {  
        InitializeComponent();  
    }  
  
    private void btn_Click(object sender, RoutedEventArgs e)  
    {  
        int num;  
        if (int.TryParse(textBox1.Text, out num))  
        {  
            lblResult.Content = $"The square of {num} is {(num * num)}";  
        }  
        else  
        {  
            lblResult.Content = "Try a number!";  
        }  
    }  
}
```

if the value in TextBox1.Text is an integer then TryParse will return true and num will contain the integer.|

## MODULE 2.

Another example of an OUT parameter. The GetFiles function returns the number of files and the OUT parameter called *fileNames* will contain an array with all the file names.

```
static void Main(string[] args)
{
    ArrayList x;

    int filecount=GetFiles(@"C:\Windows\system32", out x);

    Console.WriteLine($"The number of files in C:\\Windows\\system32 is {filecount}. Press enter to see them all.");
    Console.ReadLine();

    foreach (var item in x)
    {
        Console.WriteLine(item);
    }

    Console.WriteLine("OK. Done. Press enter to finish");

    Console.ReadLine();
}

static int GetFiles(string path, out ArrayList fileNames)
{
    int fileCount = 0;
    fileNames = new ArrayList();
    foreach (var item in Directory.GetFiles(path))
    {
        fileCount++;
        fileNames.Add(item);
    } ≤ 1ms elapsed
}

► return fileCount;
}
```

Activate ^

Ctrl + Shift + F12

## MODULE 2.

This example shows the basics of implementing exception handling

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 3;
            x = x / x;
            File.WriteAllText(@"c:\temp\result.txt", $"the result is {x}");
        }
        catch (DivideByZeroException ex)
        {

            Console.WriteLine("Sadly. the application failed because of a number. It will now close");
            Console.WriteLine("Message:" + ex.Message);
            Console.WriteLine("Stack trace:" + ex.StackTrace);
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine("The application is trying to write out to a file, but you don't have access");
            Console.WriteLine("details:" + ex.Message);
        }
        finally
        {
            Console.WriteLine("\n\nPress enter to end");
            Console.ReadLine();
        }
    }
}
```



Diagram illustrating exception handling:

- A red line points from the `DivideByZeroException` variable in the first `catch` block to the text "this variable points to an exception object".
- A blue line points from the `UnauthorizedAccessException` variable in the second `catch` block to the text "can access details about the exception".
- A blue line points from the `finally` block to the text "best practice is to catch exceptions that you expect".

this variable  
points to an  
exception object

can access details  
about the exception

best practice is to catch exceptions  
that you expect

## MODULE 2.

One Situation you want to avoid is when your application gets an exception and you can't get around it and you don't inform the user that something has gone wrong. The application might continue in a state where the data is corrupt for example.

```
STATIC INT DIVIDE(INT top, INT bottom)
{
    try
    {
        return top / bottom;
    }
    catch (Exception)
    {
    }
}
```

This is bad coding.  
It swallows the exception.  
I.e. the user might not realise something has gone wrong.|



Realise

## MODULE 2.

Sometimes you want to throw exceptions in your code because you have a situation where an application shouldn't continue. In this case an invoice is being raised with a negative amount; that's not allowed.

```
static void Main(string[] args)
{
    try
    {
        string invoice;
        invoice = ConstructInvoice(3, -6);
        Console.WriteLine(invoice);
    }
    catch (ApplicationException ex)
    {
        Console.WriteLine(ex.Message);
    }
    Console.ReadLine();
}

static string ConstructInvoice(decimal qty, decimal price)
{
    decimal invoiceTotal = qty * price;

    //Check the invoice total
    if (invoiceTotal <= 0)
        throw new ApplicationException("Invoice total is invalid. Must be >= 0");

    return $"invoice: Total Amount {invoiceTotal:C}";
}
```

here we are checking the invoice for validity

It's not valid, so pass an exception back to the caller

## MODULE 3.

This is an example of using a constructor with a Struct.

```
class Program
{
    static void Main(string[] args)
    {
        Coffee c1 = new Coffee();
        c1.BeanType = "Aribica";

        Coffee c2 = new Coffee();
        c2.BeanType = "Kenyan";

        Coffee c3 = new Coffee("PNG");
    }
}

struct Coffee
{
    public Coffee(string bType)
    {
        ≤ 1ms elapsed
        bType | ↗ "PNG" ←
        BeanType = bType;
    }
    public string BeanType;
}
```

The diagram illustrates the flow of control in the code. A red line starts at the assignment statement `Coffee c3 = new Coffee("PNG");`, points to the constructor call `new Coffee()`, and then follows the line of the constructor definition `public Coffee(string bType)`. The line then branches into the constructor's body, specifically the assignment `BeanType = bType;`. A tooltip for the variable `bType` is shown, containing the value `"PNG"`.

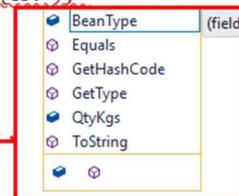
## MODULE 3.

Private fields are not visible outside a struct or class

```
Coffee c4 = new Coffee("Brazil");

Console.WriteLine($"{c3.BeanType} has a quantity of {c3.QtyKgs}, strength is: {c3._strength}");
Console.ReadLine();
}

struct Coffee
{
    public Coffee(string bType, int qty, int strength)
    {
        BeanType = bType;
        QtyKgs = qty;
        _strength = strength;
    }
    public Coffee(string bType)
    {
        BeanType = bType;
        QtyKgs = 0;
        _strength = 0;
    }
    public string BeanType;
    public int QtyKgs;
    private int _strength;
}
```



not visible because it's private

## MODULE 3.

This is an example of protecting a private field by using a Property.

```
22
23
24     c3.Strength = 12; // Line 24 highlighted with a red box
25     Console.WriteLine($"{c3.BeanType} has a quantity of {c3.QtyKgs}, strength is: {c3.Strength}");
26
27     Console.ReadLine();
28 }
29
30
31 struct Coffee
32 {
33     public Coffee(string bType, int qty, int strength)
34     {
35         BeanType = bType;
36         QtyKgs = qty;
37         _strength = strength; // Line 37 highlighted with a red circle
38     }
39     public Coffee(string bType)
40     {
41         BeanType = bType;
42         QtyKgs = 0;
43         _strength = 0;
44     }
45     public string BeanType;
46     public int QtyKgs;
47     private int _strength;
48     public int Strength
49     {
50         get { return _strength; }
51         set
52         {
53             _strength = value; // Line 53 highlighted with a red box
54         }
55     }
}
```

The reason you'd define an indexer is to make it easier for other people who use your class or struct. Indexers are special methods that use a *subscript* to set or get data. The method name for an indexer is always *this*.

- Use the **this** keyword to declare an indexer
- Use **get** and **set** accessors to provide access to the collection

```
public int this[int index]
{
    get { return this.beverages[index]; }
    set { this.beverages[index] = value; }
}
```

- Use the instance name to interact with the indexer

```
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
```

## MODULE 3.

From your notes. A simple arraylist.

```
// Create a new ArrayList object.  
ArrayList al = new ArrayList();  
// Add values to the ArrayList collection.  
0 al.Add("Value");  
al.Add("Value 2");  
1 al.Add("Value 3");  
2 al.Add("Value 4");  
  
// Remove a specific object from the ArrayList collection.  
al.Remove("Value 2"); // Removes "Value 2"  
  
// Remove an object from a specified index.  
al.RemoveAt(2); // Removes Value 4  
  
// Retrieve an object from a specified index.  
string valueFromCollection = (string)al[1]; // Returns "Value 3"
```

Every element in an *ArrayList* has the data type *object*. When you retrieve an item from an *ArrayList* it is returned as an *object*. That means you need to *Cast* (*Convert*) it to the *Data type* of the receiving variable.

## Using List Collections

page 3-13

- Add objects of any type

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
ArrayList beverages = new ArrayList();
beverages.Add(coffee1);
```

what and why is that?

- Retrieve items by index

```
Coffee firstCoffee = (Coffee)beverages[0];
```

MODULE 3:

Every element in an *ArrayList* has the data type *object*. When you retrieve an item from an *ArrayList* it is returned as an *object*. That means you need to *Cast* (*Convert*) it to the Data type of the receiving variable.

```
class Program
{
    static void Main(string[] args)
    {
        ArrayList items = ... ArrayList();
        items.Add(3);
        int result;
    }
}
```

converts the 3 to an object!

int ArrayList.Add(object value)  
Adds an object to the end of the ArrayList.  
Exceptions:  
NotSupportedException

compile. why not?

- Create a delegate for the event

```
public delegate void OutOfBeansHandler(Coffee coffee,  
EventArgs args);
```

- Create the event and specify the delegate

```
public event OutOfBeansHandler OutOfBeans;
```

A delegate is a data type that can point to a function.]

## MODULE 3.

Matches my demo 00g-AbsoluteBasicsDelegate. A delegate is a data type that can point to a method.

The screenshot shows a C# code editor with a file named Program.cs. The code defines a delegate MyDelegate that takes an int parameter and returns an int. It then uses this delegate to call a static method Cube, which calculates the cube of a given integer. The output window shows two executions of the program, both printing 'The cube of 45 is 91125'.

```
Program.cs
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace _00_AbsoluteBasicsDelegate
8  {
9      class Program
10     {
11         delegate int MyDelegate(int val);
12
13         static void Main(string[] args)
14         {
15             int y = 45;
16             Console.WriteLine($"The cube of {y} is {Cube(y)}");
17
18             MyDelegate m = Cube;
19             Console.WriteLine($"The cube of {y} is {m(y)}");
20
21
22             Console.ReadLine();
23         }
24
25         static int Cube(int x)
26         {
27             return x * x * x;
28         }
29     }
30 }
31
32
```

Annotations in red:

- 'this is a data type that I have defined.'
- 'this delegate can point to any method that accepts 1 int and returns an int.'
- 'I declare a variable called 'm' of type 'MyDelegate' and assign it 'Cube'
- 'I call Cube by using my delegate variable.'

Output window:

```
C:\Users\Admin\Desktop\MarksFiles\MarksCode\Module-03-Basic CSharp Constructs\Demos\00-AbsoluteBasicsDelegate\00-AbsoluteBasic
The cube of 45 is 91125
The cube of 45 is 91125
```

## MODULE 3.

Matches my demo *00j-UsingADelegate*. The reason for implementing a delegate in this example is so the console application can get feedback while the *ProcessFiles* method is executing.

File Edit View Project Build Debug Team Tools Test Analyze Window Help Full Screen

Class1.cs\* FileUtility Util ProcessFiles(string path)

```
7 using System;
8
9 namespace FileUtility
10 {
11     public class Util
12     {
13         public delegate void FeedbackMessage(string message);
14
15         public FeedbackMessage pointerToFeedback;
16
17         /// <summary>
18         /// Accepts a file path and returns the total bytes found for all files
19         /// </summary>
20         /// <param name="path">the file path</param>
21         /// <returns>the total bytes</returns>
22         public long ProcessFiles(string path)
23         {
24             long totalBytesFound = 0;
25             foreach (var file in Directory.GetFiles(path))
26             {
27                 if (pointerToFeedback != null)
28                     pointerToFeedback($"Processing file: {file}");
29                 totalBytesFound += file.Length;
30                 System.Threading.Thread.Sleep(20);
31             }
32             return totalBytesFound;
33         }
34     }
35 }
36 
```

Program.cs\* 00j-UsingADelegate 00j\_UsingADelegate.Program Main(string[] args)

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace _00j_UsingADelegate
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            FileUtility.Util util = new FileUtility.Util();
14            util.pointerToFeedback = ProcessFeedback;
15
16            long result = util.ProcessFiles(@"c:\windows\system32");
17
18            Console.WriteLine($"The total bytes found is {result} ");
19
20            Console.ReadLine();
21        }
22
23        static void ProcessFeedback(string msg)
24        {
25            Console.WriteLine(msg);
26        }
27    }
28 }
29 
```

here the method is called using the delegate variable

this sets the delegate variable to point to a function

this method matches the delegate, i.e. returns void and accepts a string

Output Ready

Ln 28 Col 51 Ch 51 INS ↑ 0 ↗ 2 MarksFiles main

## MODULE 3.

Events extend the idea of delegates and make things easier for the caller of a struct or class. Line 13 (left) is the extra key line I added to define an event. Then, line 14 (right) connects the event to a method. When line 29 (left) runs the method at line 24 (right) will execute.

```
12     public delegate void FeedbackMessage(string message);
13     public event FeedbackMessage FeedbackEvent;
14
15     public FeedbackMessage pointerToFeedback;
16
17
18     /// <summary>
19     /// Accepts a file path and returns the total bytes found for all fi
20     /// </summary>
21     /// <param name="path">the file path</param>
22     /// <returns>the total bytes</returns>
23     public long ProcessFiles(string path)
24     {
25         long totalBytesFound = 0;
26         foreach (var file in Directory.GetFiles(path))
27         {
28             if (FeedbackEvent != null)
29                 FeedbackEvent($"Processing file: {file}");
30             totalBytesFound += file.Length;
31             System.Threading.Thread.Sleep(20);
32         }
33         return totalBytesFound;
34     }
35
36 }
37
```

```
2     using System.Collections.Generic;
3     using System.Linq;
4     using System.Text;
5     using System.Threading.Tasks;
6
7     namespace _00_FromDelegates to Events
8     {
8.5         class Program
9         {
10            static void Main(string[] args)
11            {
12                FileUtility.Util util = new FileUtility.Util();
13                util.FeedbackEvent += Util_FeedbackEvent;
14
15
16                long result = util.ProcessFiles(@"c:\windows\system32");
17
18                Console.WriteLine($"The total bytes found is {result} ");
19
20                Console.ReadLine();
21
22
23
24            private static void Util_FeedbackEvent(string message)
25            {
26                Console.WriteLine(message);
27            }
28        }
29    }
```

## MODULE 3.

I included this example to show how a delegate and event was used in practice

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9 using System.Windows.Forms;
0
1 namespace WindowsFormsApp1
2 {
3     public partial class Form1 : Form
4     {
5         public Form1()
6         {
7             InitializeComponent();
8             button1.Click += Button1_Click;
9         }
10        private void Button1_Click(object sender, EventArgs e)
11        {
12            MessageBox.Show("hello");
13        }
14    }
}
```

It's easy to setup the event for me.  
Autocomplete helps me.

This button class was written in 2001. The person who wrote it had no idea that it would call my method

Press this

## MODULE 4

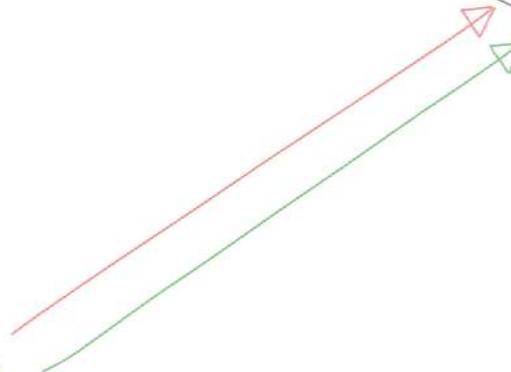
What is object oriented software?  
Why do we have it?



### User Story (requirements)

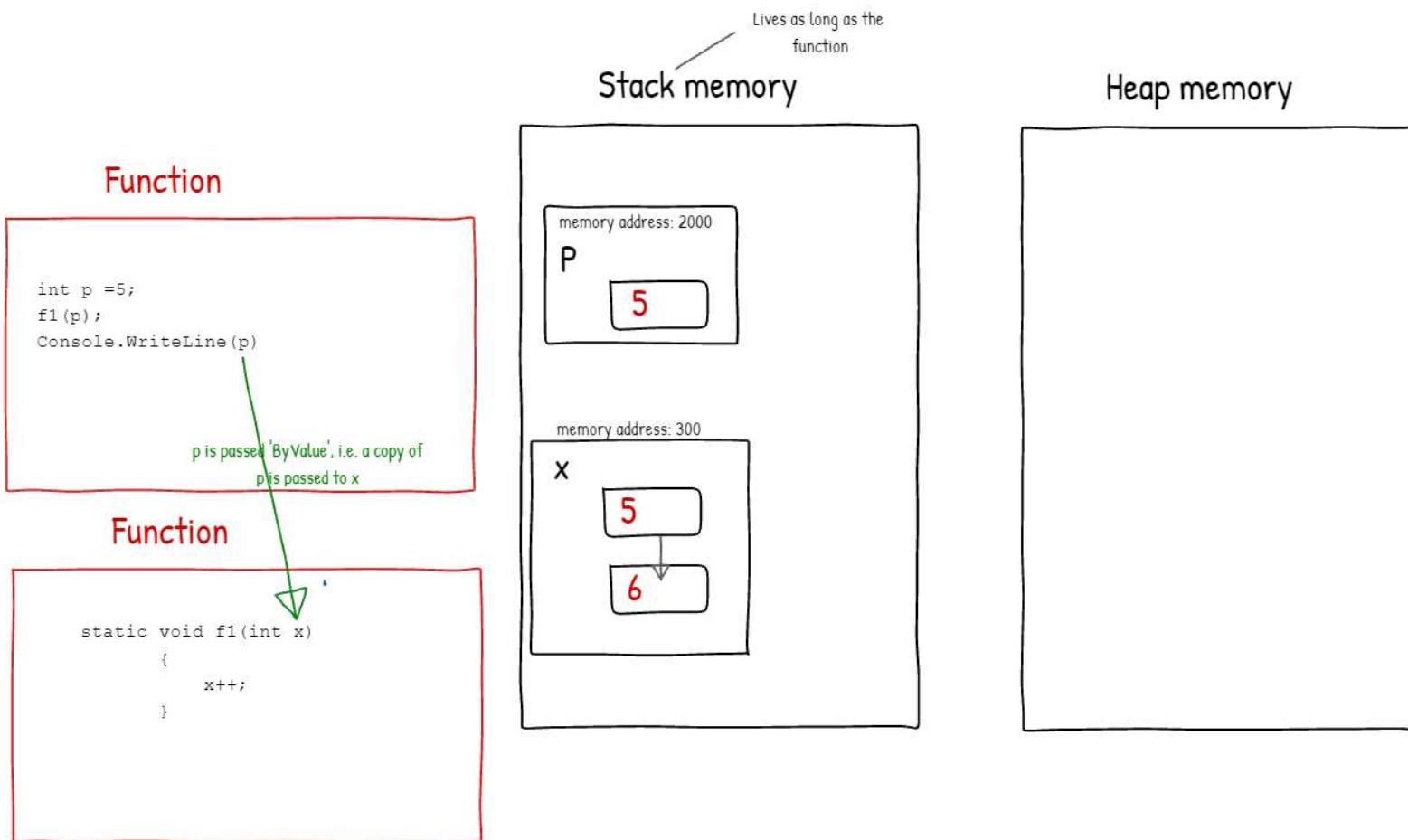
A customer can open several types of bank accounts. Every Bank Account has an account number, a balance, supports deposit and withdraw

Structs, Classes, Properties,  
Methods, Fields



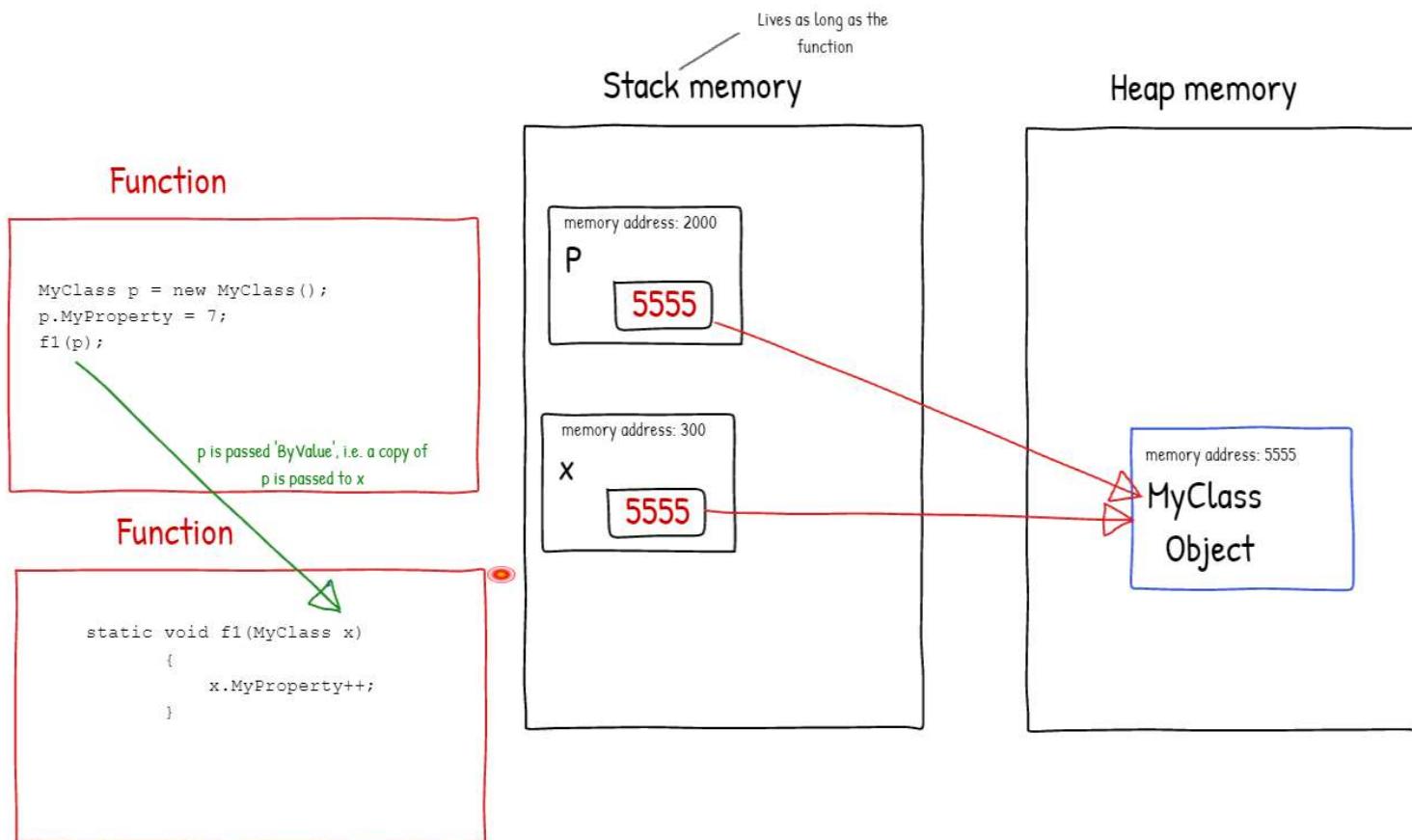
## MODULE 4.

When you pass parameters to functions they are passed **ByValue** by default. That means a copy of the stack variable is passed to the parameter of the receiving function. If the receiving function changes the parameter it doesn't affect the stack variable that's passed. Note that in this example the datatype being used is an int which is a Struct.



## MODULE 4.

When you pass parameters to functions they are passed **ByValue** by default. That means a copy of the stack variable is passed to the parameter of the receiving function. If the receiving function changes the parameter it doesn't affect the stack variable that's passed. Note that in this example the datatype being used is **MyClass** which is a **Class**. You want to be extra careful here, the function **f1** does not change the parameter **x**, but does change a property of the object that **x** points to, which is the same object that **p** points to.



## MODULE 4.

Interfaces are contracts. They describe a set of functionality that a class guarantees to implement. In this example there is an interface that two bank account classes implement which includes a deposit and withdraw function.

```
BankAccount first = new BankAccount();
first.AccountNumber = "111";
first.Deposit(200);

BankAccount second = new BankAccount();
second.AccountNumber = "222";
second.Deposit(100);

Transfer(first, second,20);

Console.WriteLine($"After, Bank Account {first.AccountNumber}, Balance {first.Balance:C}");
Console.WriteLine($"After, Bank Account {second.AccountNumber}, Balance {second.Balance:C}");

Console.ReadLine();
}

static void Transfer(IBank from, IBank To, decimal amount)
{
    from.Withdraw(amount);
    To.Deposit(amount);
}

interface IBank
{
    void Deposit(decimal amount);
    void Withdraw(decimal amount);
}

class BankAccount : IBank
{
    public string AccountNumber { get; set; } //property

    private decimal _balance; //field
    public decimal Balance... //property

    public void Deposit(decimal amount) //Method
    {
        _balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (_balance >= amount)
            _balance -= amount;
        else
            Console.WriteLine("Insufficient balance");
    }
}
```



I can pass any objects here  
that implement IBank

this class implements IBank interface

It must implement deposit and withdraw

## MODULE 4.

Interfaces can be defined in a different file to the class that implements them

```
namespace _00e_BasicInterfaces
{
    class Program
    {
        static void Main(string[] args)...
        static void Transfer(IBank from, IBank To, decimal amount)...
    }
    interface IBank...
    class BankAccount ...
}
```

These could be in 3  
different files.|

## MODULE 4.

From my slides about generics. The code in the centre of the page below doesn't compile. Why not?

# Generics DEMO 6

## Generics and Constraints

What's wrong here?

string a="zz";  
string b="gg";

string c = a \* b;

string does  
not support  
this operator

```
class Testerr<T>
{
    public T test(T value)
    {
        return value;
    }

    public T ReturnTheBiggest(T x, T y)
    {
        if (x > y)
        {
            return x;
        }
        else
        {
            return y;
        }
    }
}
```

there is no  
guarantee that  
type T supports  
this operator.|

## MODULE 4.

An example of using a Generic Class that is part of the .net framework

```
Dictionary<string, int> people = new Dictionary<string, int>();  
people.Add("dave", 22);  
people.Add("Eng", 55);
```

## MODULE 4, review

```
-- 33     class BankAccount
34     {
35
36         public static decimal InterestRate { get; set; } ← property
37
38         public string AccountNumber { get; set; } //property
39
40         private decimal _balance; //field
41         public decimal Balance
42         {
43             get
44             {
45                 return _balance;
46             }
47         } //property
48
49         public void Deposit(decimal amount) //Method
50         {
51             _balance += amount;
52         }
53
54         public void Withdraw(decimal amount) //Method
55         {
56             _balance -= amount;
57         }
58
59         public decimal GetInterest()
60         {
61             return _balance * InterestRate;
62         }
63
64         public static void IncreaseInterestrate()
```

property

Method

static means:

1. It can be accessed without creating an instance.
2. It is shared by all instances

## MODULE 4, review

The screenshot shows the Microsoft Visual Studio IDE interface. On the left is the 'Program.cs' code editor window. The code defines a `BankAccount` class with static and instance members. A red box highlights the static modifier on the `InterestRate` property and the `IncreaseInterestrate()` method. Another red box highlights the static modifier on the `Balance` property. Handwritten annotations in blue ink explain the behavior of these members:

- A blue arrow points from the word "static" in the `InterestRate` declaration to the text "All BankAccounts share this data".
- A blue arrow points from the word "static" in the `IncreaseInterestrate()` declaration to the text "calling instance method".
- A blue arrow points from the word "static" in the `Balance` declaration to the text "Instance data, i.e., each BankAccount has its own value".

The code editor also shows other annotations like "100 %", "Ln 23", "Col 31", "Ch 31", "INS", and "MarksFiles". The Solution Explorer window on the right shows a single project named '00d-StaticClassMembers' with files like 'Properties', 'References', 'App.config', and 'Program.cs'.

```
22
23     BankAccount second = new BankAccount();
24     second.AccountNumber = "222";
25     second.Deposit(100);
26     second.Withdraw(10);
27     Console.WriteLine($"Bank Account {second.AccountNumber}, Balance {second.Balance:C}, Current Interest {second.GetInterest():C}");
28
29     Console.ReadLine();
30 }
31
32 class BankAccount
33 {
34
35     public static decimal InterestRate { get; set; }
36
37     public string AccountNumber { get; set; } //property
38
39     private decimal _balance; //field
40     public decimal Balance
41     {
42         get
43         {
44             return _balance;
45         }
46     } //property
47
48     public void Deposit(decimal amount) //Method
49     {
50         _balance += amount;
51     }
52
53     public void Withdraw(decimal amount) //Method
54     {
55         _balance -= amount;
56     }
57
58     public decimal GetInterest()
59     {
60         return _balance * InterestRate;
61     }
62
63     public static void IncreaseInterestrate()
64 }
```

## MODULE 4, review

The diagram shows a screenshot of Microsoft Visual Studio with a code editor displaying C# code. A red box highlights the `IBank` interface definition, which contains two methods: `Deposit` and `Withdraw`. A green box highlights the `Balance` property definition in the `BankAccount` class. A blue box highlights the implementation of the `Deposit` and `Withdraw` methods in the `BankAccount` class. A red wavy line points from the `IBank` interface to the `Balance` property. A blue wavy line points from the `IBank` interface to the `Deposit` and `Withdraw` methods. A green wavy line points from the `Balance` property to the `Deposit` and `Withdraw` methods. A blue wavy line points from the `Deposit` and `Withdraw` methods back to the `IBank` interface. The code editor shows lines 43 to 76 of the `Program.cs` file.

```
43     from.Withdraw(amount);
44     To.Deposit(amount);
45 }
46 }
47 }
48 }
49 interface IBank
50 {
51     void Deposit(decimal amount);
52     void Withdraw(decimal amount);
53     decimal Balance { get; }
54 }
55 class BankAccount : IBank
56 {
57     public string AccountNumber { get; set; } //property
58
59     private decimal _balance; //field
60     public decimal Balance
61     {
62         get
63         {
64             return _balance;
65         }
66     } //property
67
68     public void Deposit(decimal amount) //Method
69     {
70         _balance += amount;
71     }
72
73     public void Withdraw(decimal amount) //Method
74     {
75         _balance -= amount;
76     }
77 }
```

Interfaces are contracts.  
Interfaces can define methods  
Interfaces can define properties

Interfaces are 'implemented' by classes or structs.

When a class implements an interface it guarantees to implement the whole contract (i.e. all the members)

## MODULE 4, review

```
28     Transfer(first, second, 20);
29     Transfer(second, first, 10);
30
31
32     Console.WriteLine($"After, Bank Account {first.AccountNumber}, Balance {first.Balance:C}");
33     Console.WriteLine($"After, Bank Account {second.AccountNumber}, Balance {second.Balance:C}");
34
35     Console.ReadLine();
36 }
37
38 static void Transfer(IBank from, IBank To, decimal amount)
39 {
40     if (from.Balance < amount)
41         throw new Exception("insufficient Funds");
42     from.Withdraw(amount);
43     To.Deposit(amount);
44 }
45
46
47 interface IBank
48 {
49     void Deposit(decimal amount);
50     void Withdraw(decimal amount);
51     decimal Balance { get; }
52 }
53
54 class BankAccount : IBank
55 {
56     public string AccountNumber { get; set; } //property
57 }
```

This says you can transfer 'from' any object that implements IBank, and you can transfer 'to' any object that implements IBank



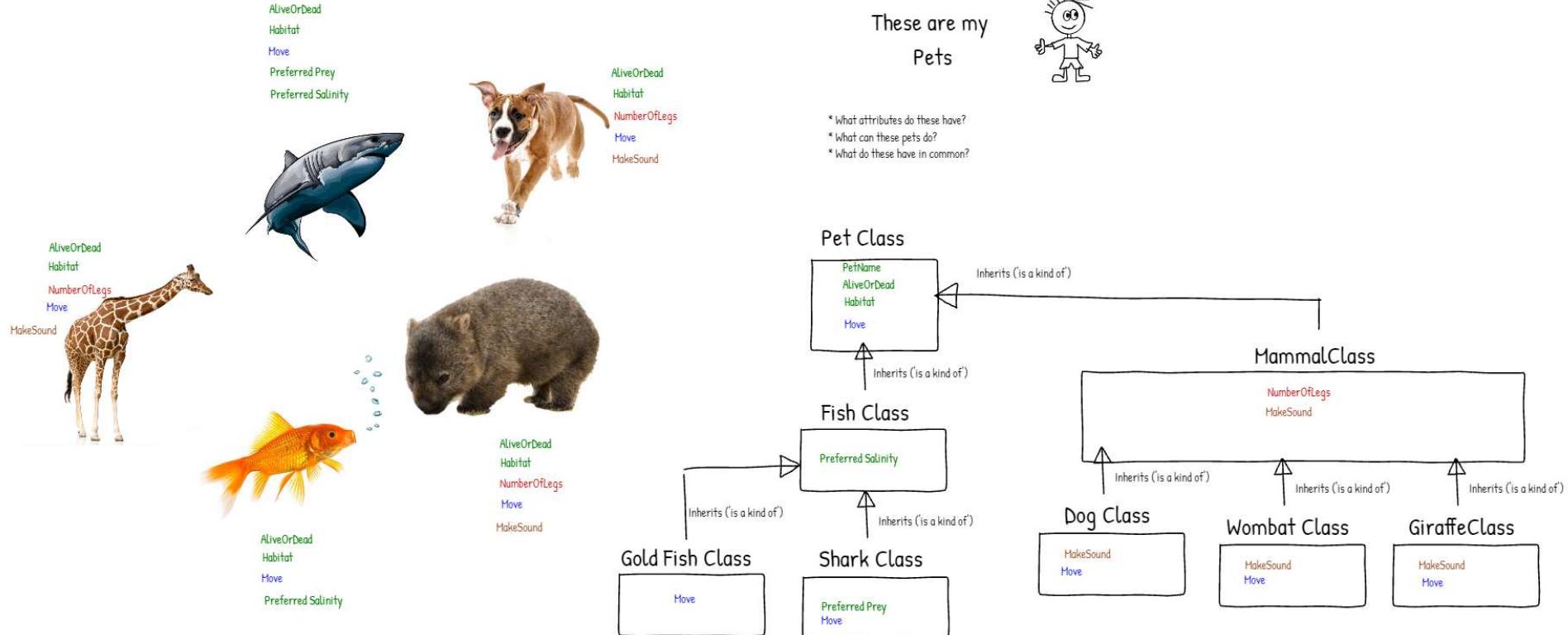
The screenshot shows a Microsoft Visual Studio interface with the following details:

- Solution Explorer:** Shows a solution named '00e-BasicInterfaces' containing a project '00e-BasicInterfaces' with files 'Properties', 'References', 'App.config', and 'Program.cs'.
- Code Editor:** Displays the 'Program.cs' file with the following code:

```
28     Transfer(first, second, 20);
29     Transfer(second, first, 10);
30
31     Console.WriteLine($"After, Bank Account {first.AccountNumber}, Balance {first.Balance:C}");
32     Console.WriteLine($"After, Bank Account {second.AccountNumber}, Balance {second.Balance:C}");
33
34     Console.ReadLine();
35 }
36
37 static void Transfer(IBank from, IBank To, decimal amount)
38 {
39     if (from.Balance < amount)
40         throw new Exception("insufficient Funds");
41     from.Withdraw(amount);
42     To.Deposit(amount);
43 }
44
45 interface IBank
46 {
47     void Deposit(decimal amount);
48     void Withdraw(decimal amount);
49     decimal Balance { get; }
50 }
51
52 class BankAccount : IBank
53 {
54     public string AccountNumber { get; set; } //property
55
56     private decimal _balance; //field
57     public decimal Balance
58     {
59
60     }
```
- Annotations:** A blue box highlights the `Transfer` method. A red box highlights the `IBank` interface. A red circle highlights the `AccountNumber` property in the `BankAccount` class. A blue arrow points from the `IBank` interface to the `Transfer` method parameters. A blue box encloses the `IBank` interface definition. A red box encloses the `BankAccount` class definition.
- Text Overlay:** A large blue annotation at the top right reads "parameter, it can point to anything that implements IBank". A large red annotation on the right side reads "This says you can transfer 'from' any object that implements IBank, and you can transfer 'to' any object that implements IBank".

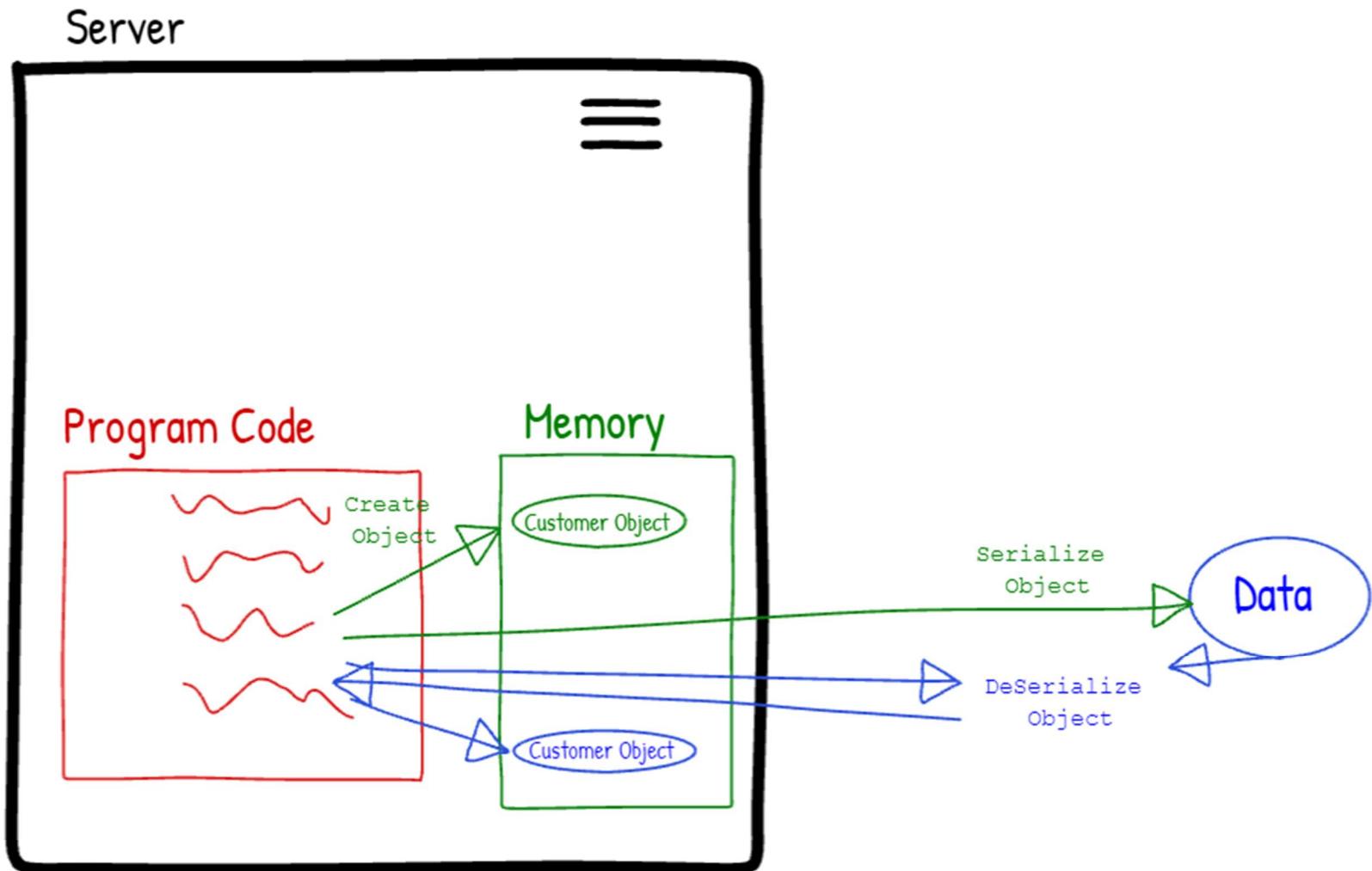
## MODULE 5

This is the diagram built during class to help understand how to define class hierarchies. It matches the code that I wrote.



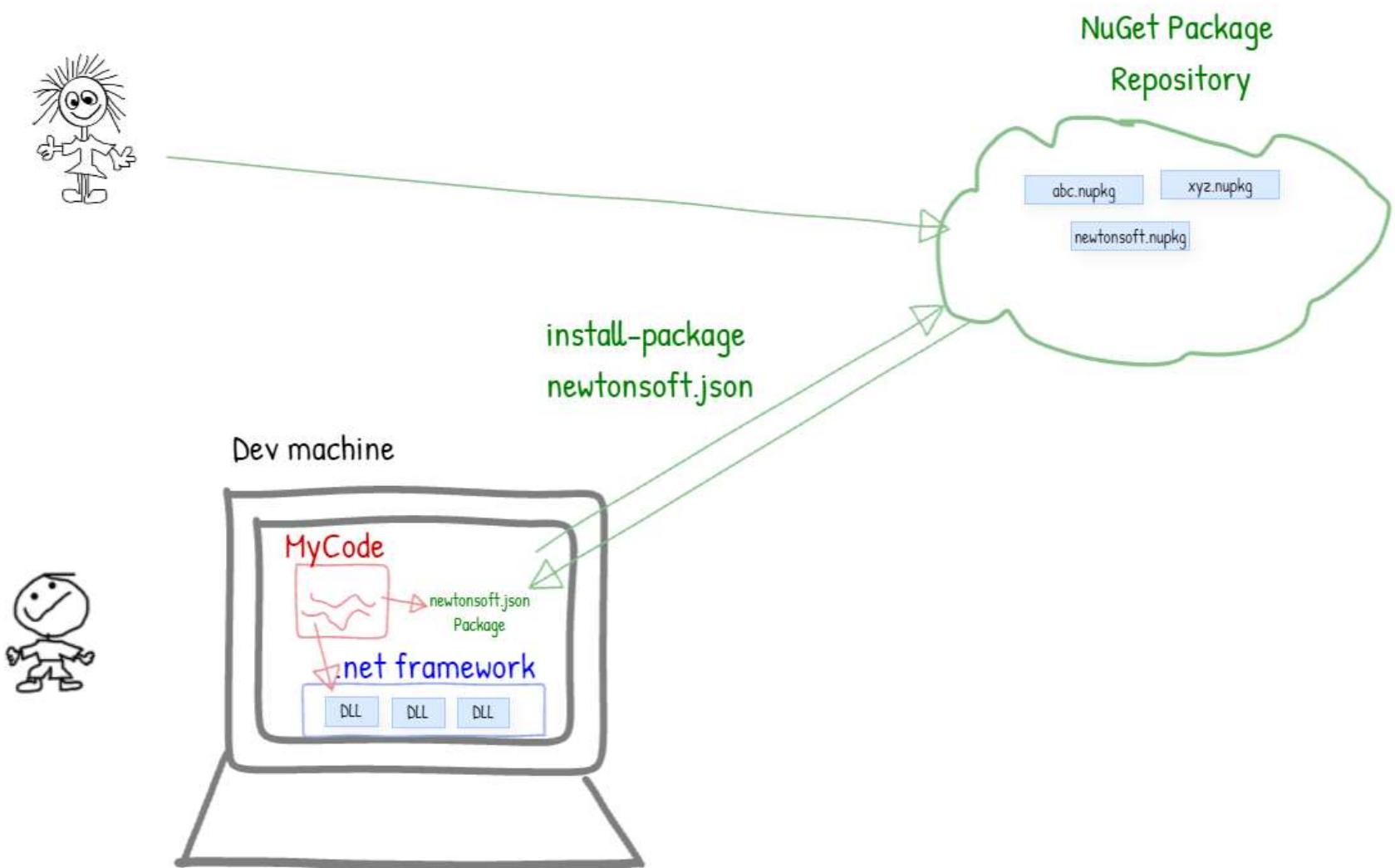
## MODULE 6.

It's often a requirement to take objects in memory and then save or send them somewhere. That process is called *serialization* and the reverse process is called *deserialization*.



## MODULE 6:

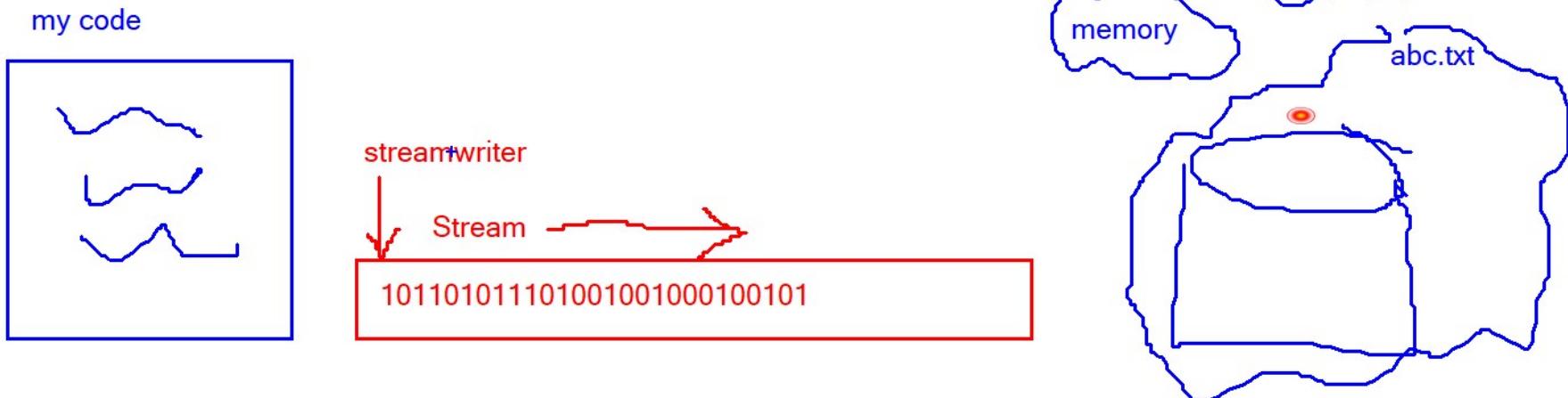
The lab in Module6 used the Newtonsoft.json NuGet Package



Notice *Stream* and *StreamWriter* in the diagram below

## Lesson 3: Performing I/O by Using Streams

- What are Streams?
- Types of Streams in the .NET Framework
- Reading and Writing Binary Data by Using Streams
- Reading and Writing Text Data by Using Streams



Before we started the module, it's a good idea to ask *What is a database?*

- Creating and Using Entity Data Models
- Querying Data by Using LINQ

"what is a database?"

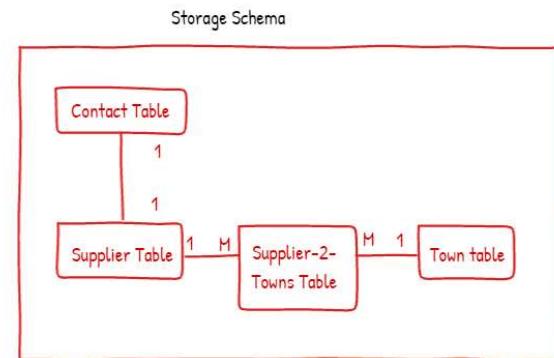
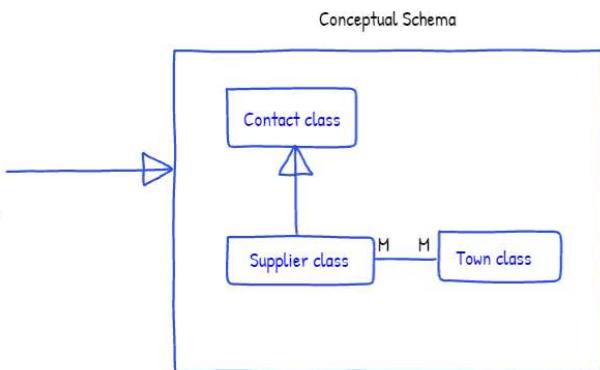
Styles

- relational
- key\value
- wide column
- network
- hierarchical

- stores data
- electronic/digital storage
- has a physical and logical architecture
- can search/query data
- supports create/read/update/delete
- has a schema storage mechanism

## MODULE 7.

Before we had the entity framework writing a C# application that performed CRUD operations on a database was cumbersome. Why? because we would need to write extensive amounts of code to move the data coming from a database into our C# conceptual schema (i.e. our C# classes). Relational database schema usually do not match the OO class structure we write in our code, so we'd have to write a whole heap of mapping code. The diagram below shows a database structure that does not match our C# classes. Relational databases don't support inheritance or many-to-many relationships.



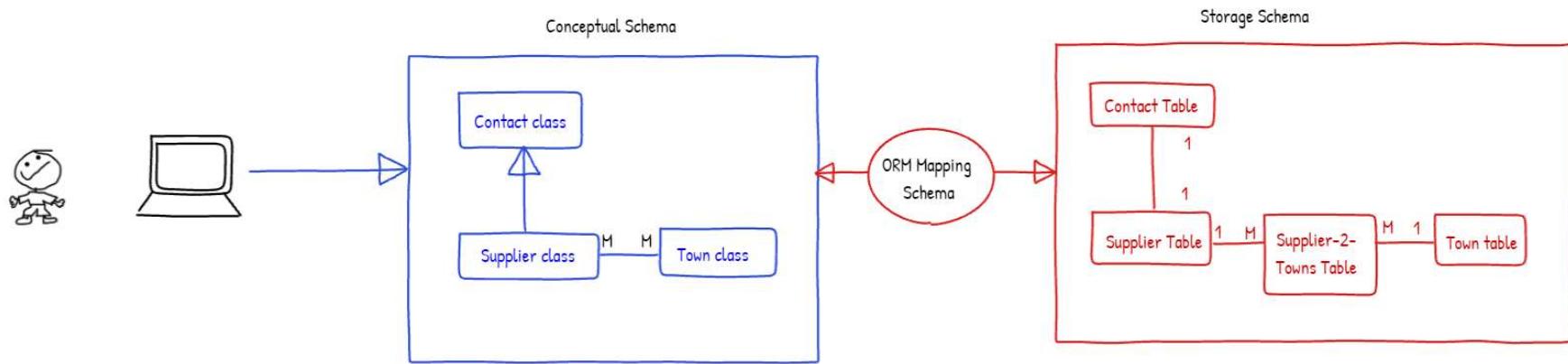
sid	s_sname
1	ACME
2	Coke

sid	tid
1	2
1	1
2	2

tid	t_name
1	Lithgow
2	Perth

## MODULE 7.

The Entity Framework provides the ORM mapping code for us so we can concentrate on writing our code against C# classes.



## MODULE 7.

When using the Entity Framework and in many other .Net programming situations you find yourself needing to use Lambda expressions. A *Lambda* expression is simply an in-line function. Button1.Click (a property) must be assigned a delegate (a data type that points to a method). In the below code that is done with a Lambda.

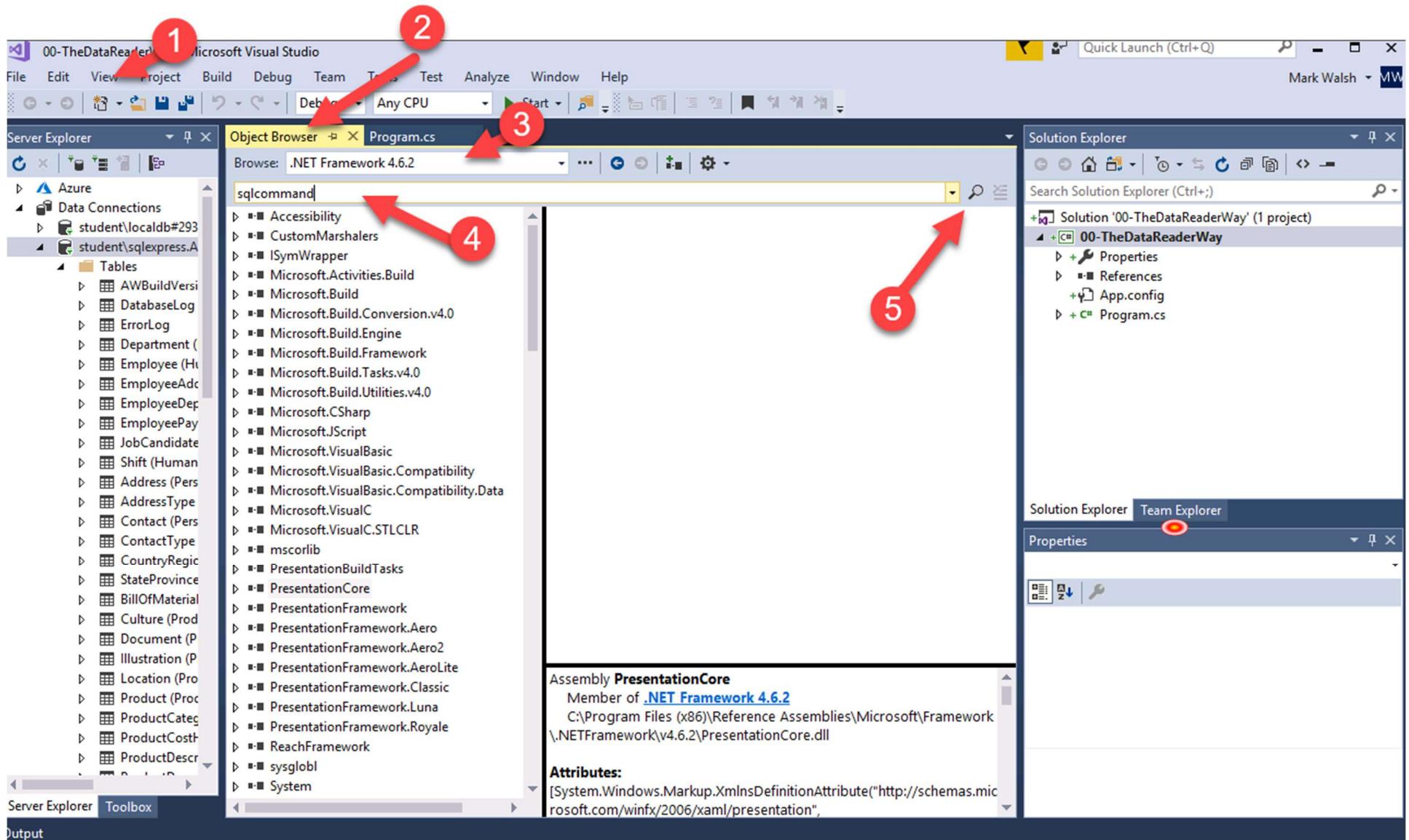
```
9  using System.Windows.Forms;                                     method
0
1  namespace WindowsFormsApp1
2  {
3      public partial class Form1 : Form
4      {
5          public Form1()
6          {
7              InitializeComponent();
8              button1.Click += ( sender, e) => MessageBox.Show("Hello " + ((Button)sender).Text);
9          }
10     }
11 }
```

The diagram highlights a portion of the C# code within a blue-bordered box. Inside this box, several red annotations are present:

- A red arrow points from the text "method parameters" to the parameters `( sender, e)` in the lambda expression.
- A red arrow points from the text "method body" to the code inside the lambda expression: `=> MessageBox.Show("Hello " + ((Button)sender).Text);`.
- A red arrow points from the text "Lambda operator" to the lambda operator itself: `&gt;`.
- A small red circle with a dot is located at the bottom left of the box.

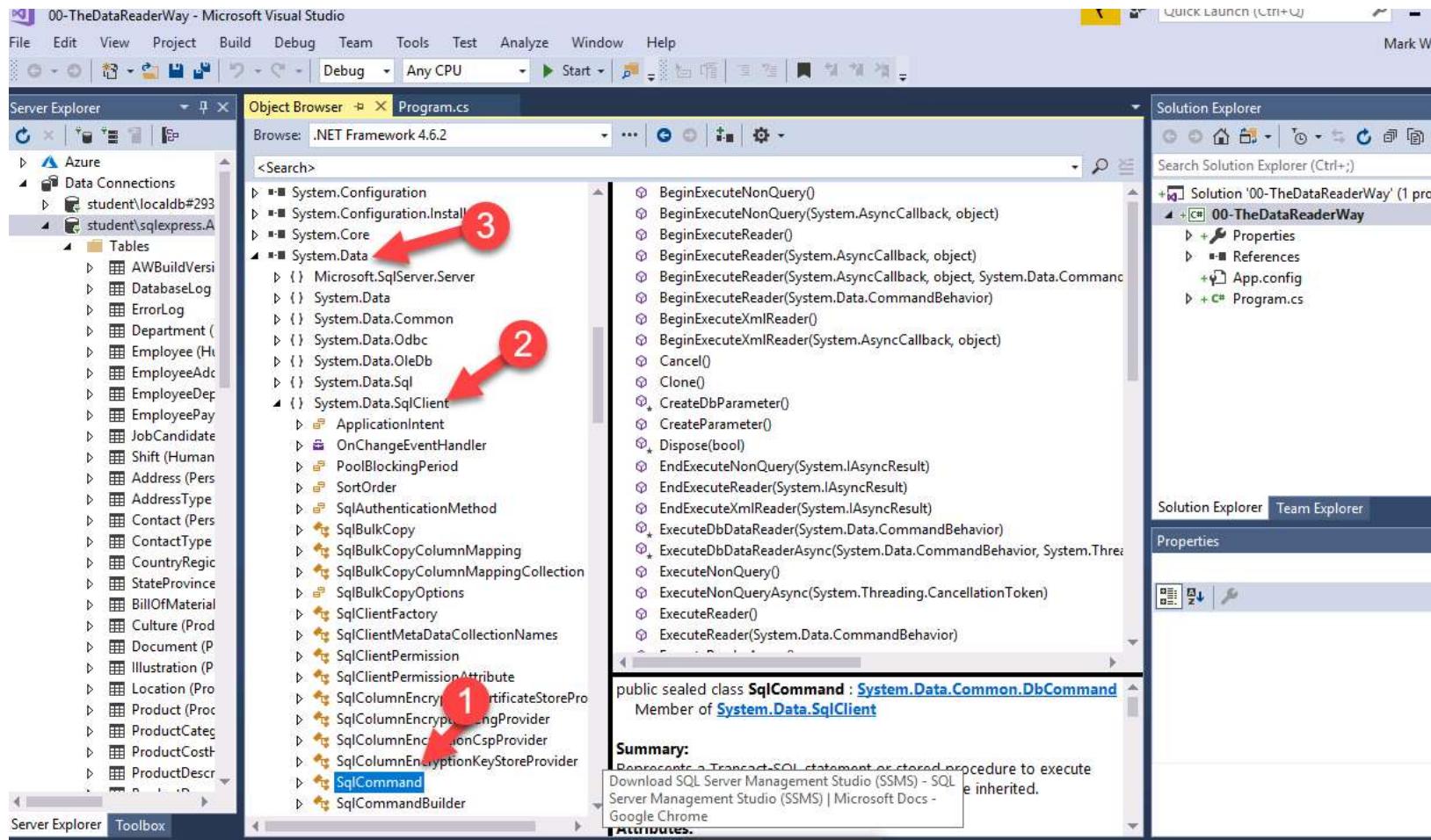
## MODULE ALL

A very useful tool in Visual Studio is the *Object Browser*. It's great for finding classes and structs and determining their parent namespaces and the DLL they reside in.



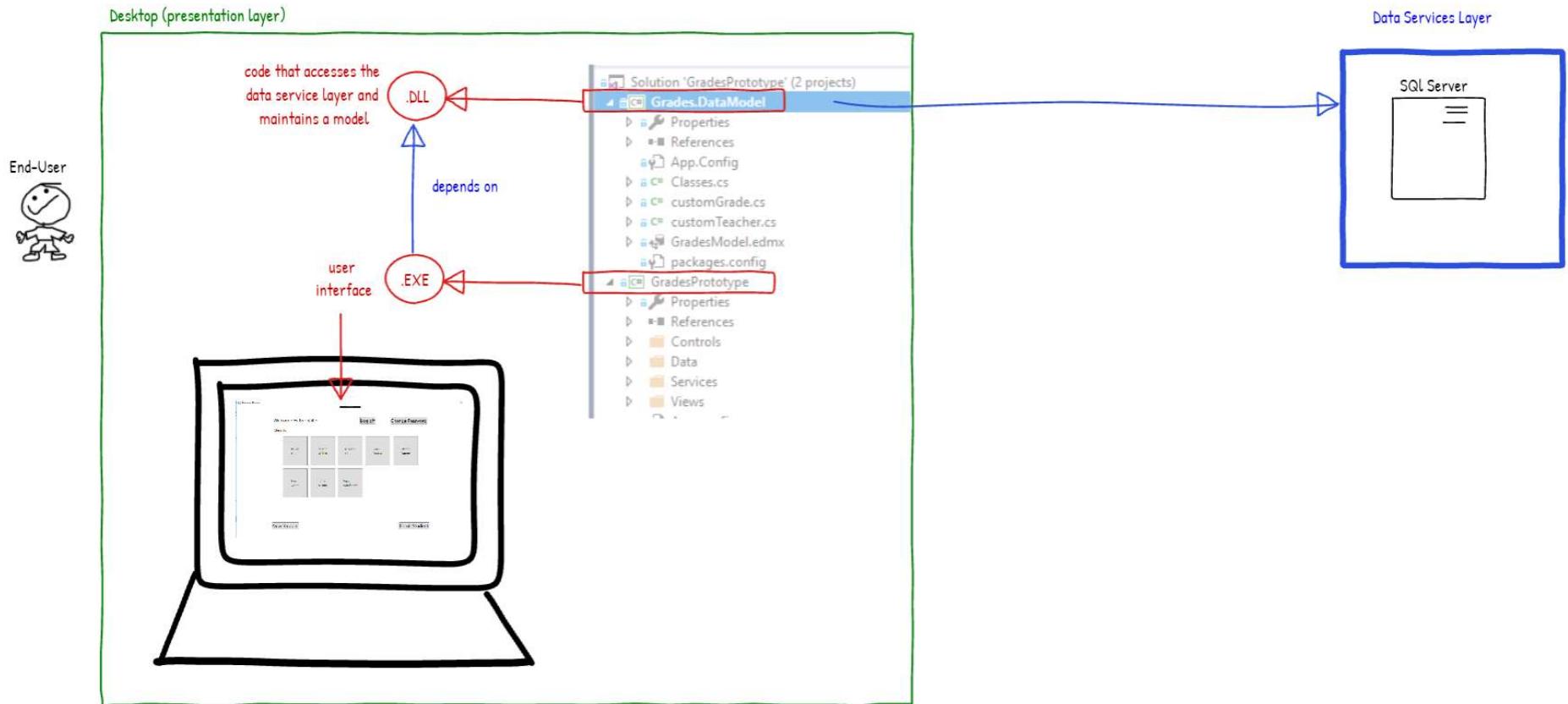
## MODULE ALL

A very useful tool in Visual Studio is the *Object Browser*. It's great for finding classes and structs and determining their parent namespaces and the DLL they reside in.



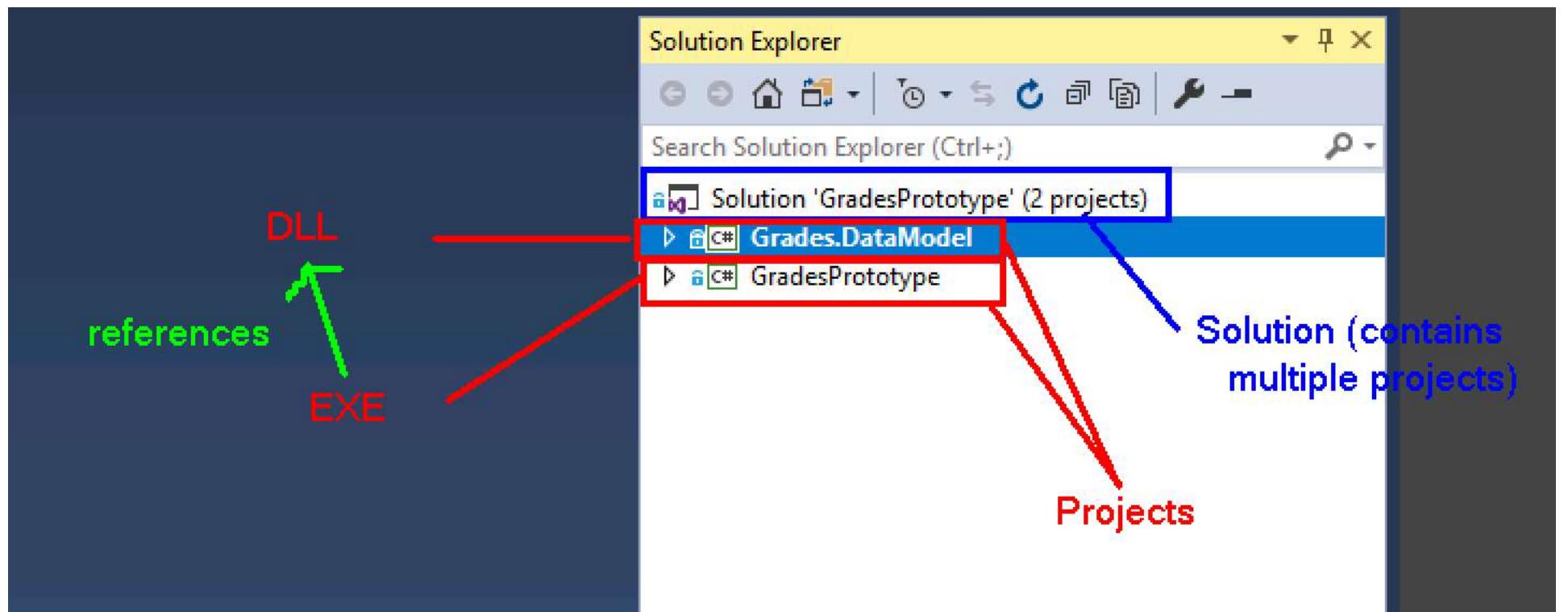
## MODULE 7

### The module 7 lab exercise architecture

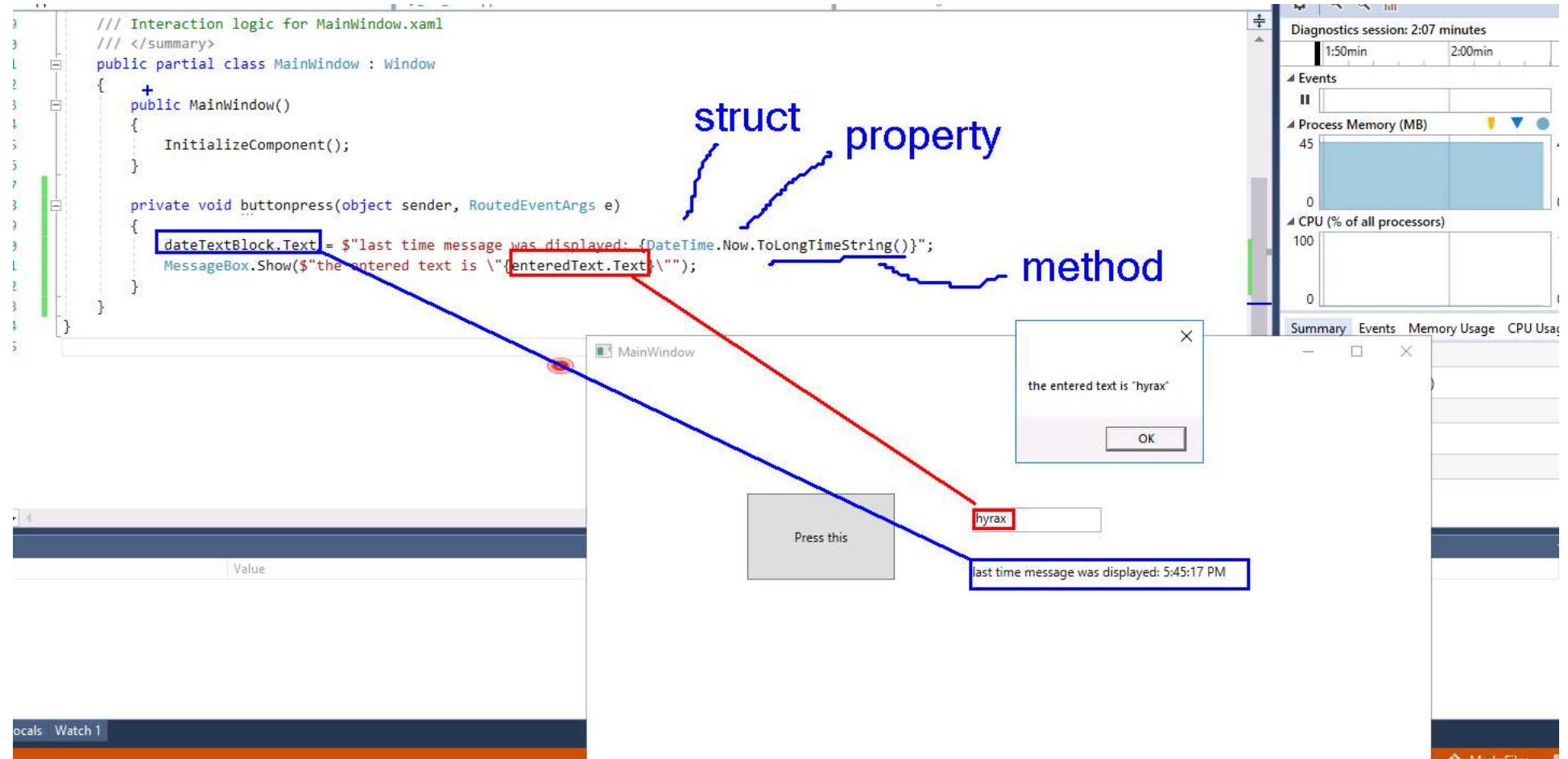


## MODULE All Modules

When you work with Visual Studio, there is a concept of a *Solution* and of *Projects*. A solution is composed of one or more projects. Each project is concerned with a certain aspect of an application such as the user interface or accessing a database.

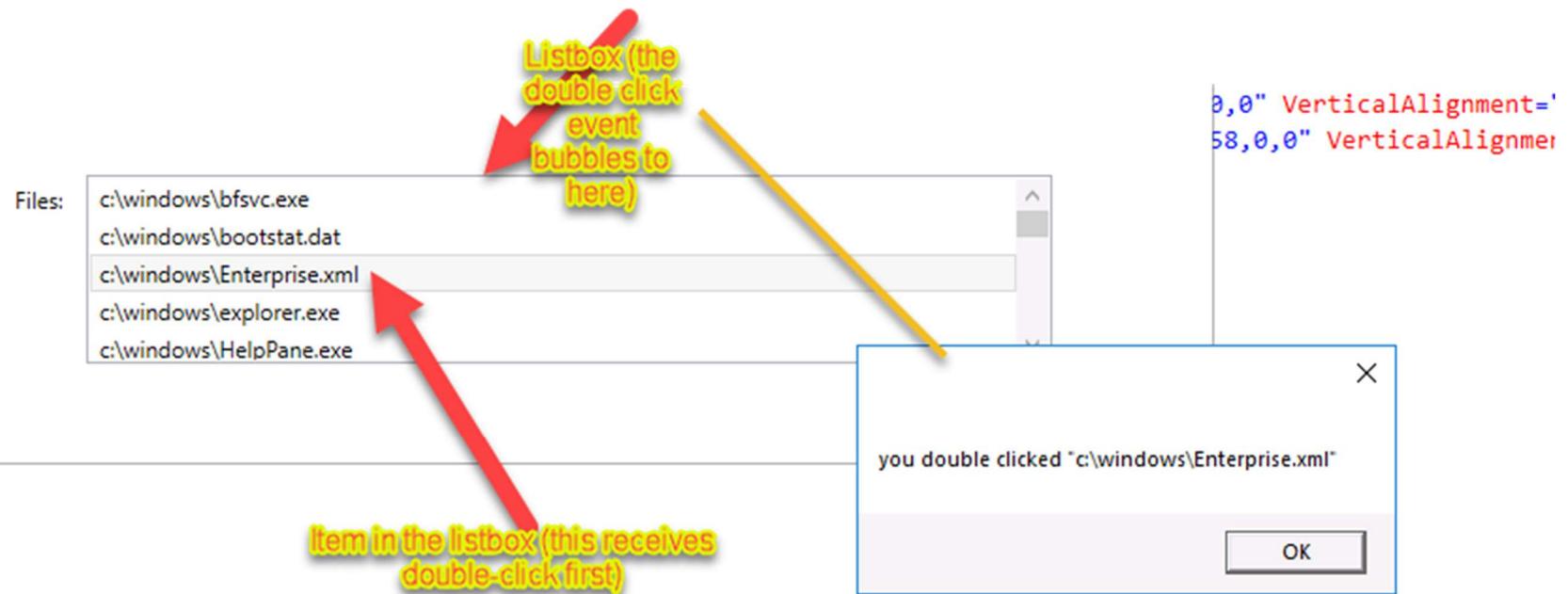


## MODULE 9



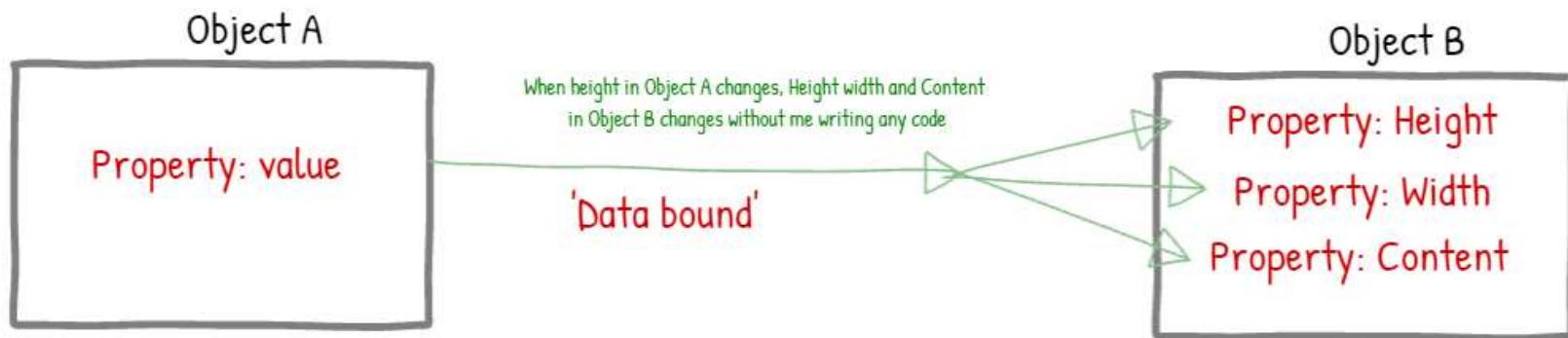
## MODULE 9

Concept of Event Bubbling in WPF.



## MODULE 9

The concept of *Data Binding*



## MODULE 9

### A data binding example

Creates an instance of a

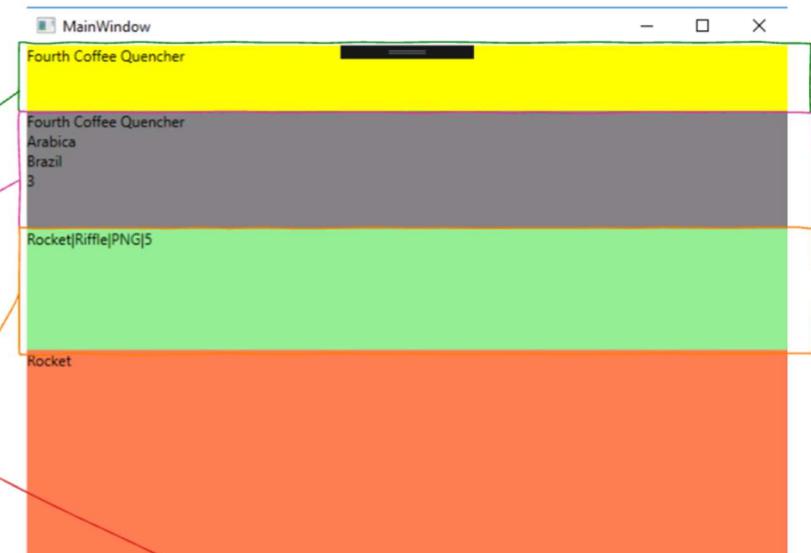
Coffee object

Examples of  
data binding

MainWindow.xaml

```
<Window.Resources>
    <local:Coffee x:Key="coffee1" Name="Fourth Coffee Quencher"
        Bean="Arabica"
        CountryOfOrigin="Brazil"
        Strength="3" /></local:Coffee>
</Window.Resources>

<Grid>
    <TextBlock Background="Yellow" Text="{Binding Source={StaticResource coffee1}, Path=Name}" Margin="0,0,0,0">
        <StackPanel Background="Gray" Margin="0,53,0,194">
            <StackPanel.DataContext>
                <Binding Source={StaticResource coffee1}" />
            </StackPanel.DataContext>
            <TextBlock Text="{Binding Path=Name}" />
            <TextBlock Text="{Binding Path=Bean}" />
            <TextBlock Text="{Binding Path=CountryOfOrigin}" />
            <TextBlock Text="{Binding Path=Strength}" />
        </StackPanel>
    </TextBlock>
    <StackPanel Background="LightGreen" x:Name="SPH" Margin="0,148,0,99" Orientation="Horizontal">
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock Text="|"/>
        <TextBlock Text="{Binding Path=Bean}" />
        <TextBlock Text="|"/>
        <TextBlock Text="{Binding Path=CountryOfOrigin}" />
        <TextBlock Text="|"/>
        <TextBlock Text="{Binding Path=Strength}" />
    </StackPanel>
    <TextBlock Background="Coral" x:Name="TB" Margin="0,246,0,32" />
</Grid>
```



Classes.cs

```
public class Coffee
{
    public string Name { get; set; }
    public string Bean { get; set; }
    public string CountryOfOrigin { get; set; }
    public string Strength { get; set; }
}
```

## MODULE 9

A data binding example using a ListBox item template.

The screenshot illustrates a Windows application window titled "MainWindow". Inside the window, there is a `ListBox` control named `cflist`. The `ItemTemplate` for this `ListBox` is defined as a `Grid` with four rows. The first row (`Grid.Row="0"`) contains a `TextBlock` with `FontSize="22"`, `Background="Black"`, and `Foreground="White"`, displaying the value "Rocket". The subsequent three rows (`Grid.Row="1"`, `Grid.Row="2"`, `Grid.Row="3"`) each contain a `TextBlock` with `FontSize="14"` and `Background="White"`, displaying the values "VotingBox", "PNG", and "5" respectively. The application's title bar shows the name "J".

Annotations in the image:

- "Each item in the listbox will be a grid": Points to the `ItemTemplate` section of the XAML code.
- "Grid has 4 rows": Points to the `RowDefinitions` section of the XAML code.
- "This is the format for each item in the listbox": Points to the `DataTemplate` section of the XAML code.
- "One listbox Item": Points to the `ListBox` control in the application window.

XAML Code:

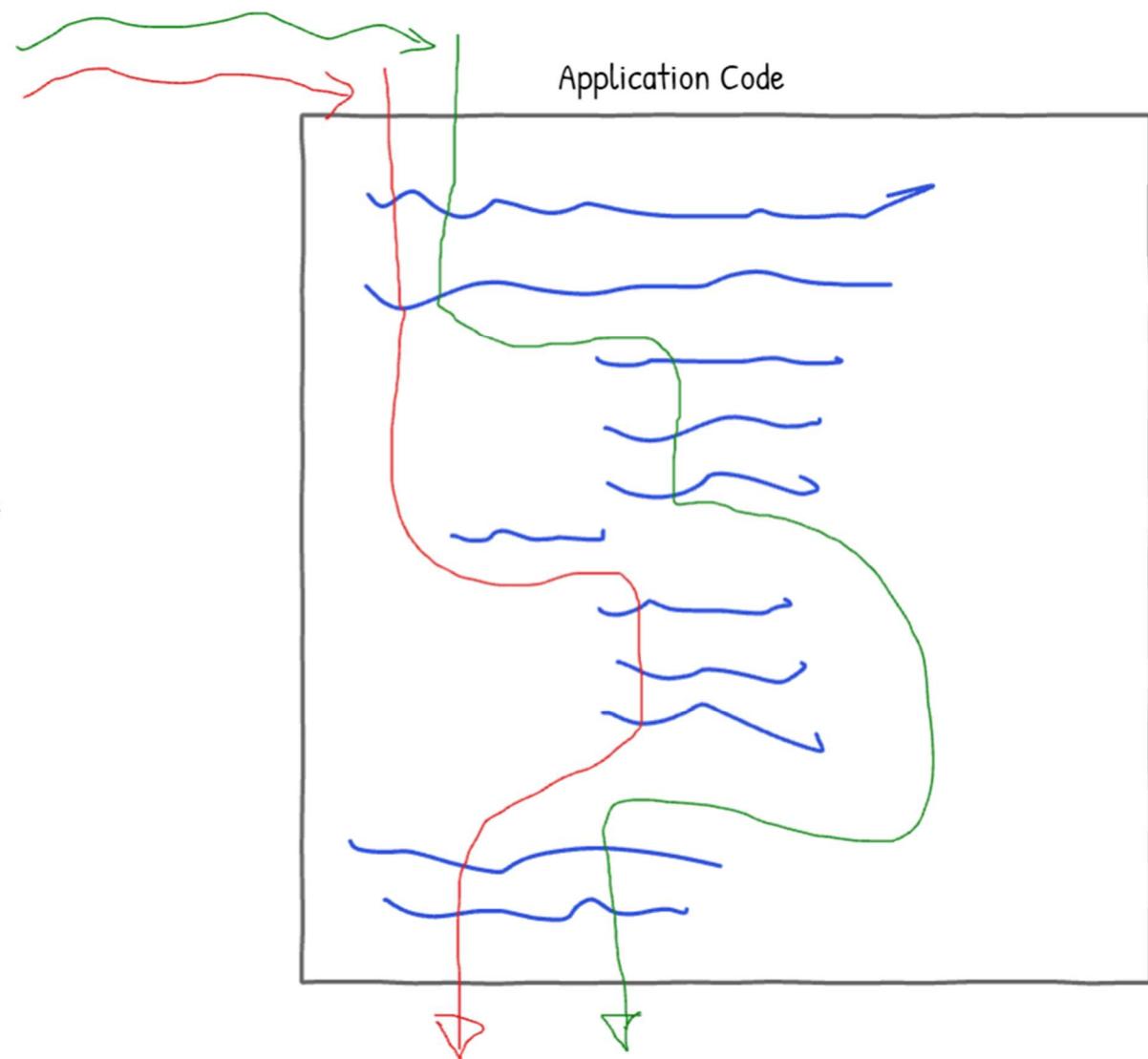
```
<Grid>
    <ListBox x:Name="cflist" Margin="30,29,144,43">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Grid.RowDefinitions>
                        <RowDefinition Height="2*" />
                        <RowDefinition Height="*" />
                        <RowDefinition Height="*" />
                        <RowDefinition Height="*" />
                    </Grid.RowDefinitions>
                    <TextBlock Text="{Binding Path=Name}" Grid.Row="0" FontSize="22" Background="Black" Foreground="White" />
                    <TextBlock Text="{Binding Path=Bean}" Grid.Row="1" />
                    <TextBlock Text="{Binding Path=CountryOfOrigin}" Grid.Row="2" />
                    <TextBlock Text="{Binding Path=Strength}" Grid.Row="3" />
                </Grid>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
```

## MODULE 10

### What is Multi-threading?

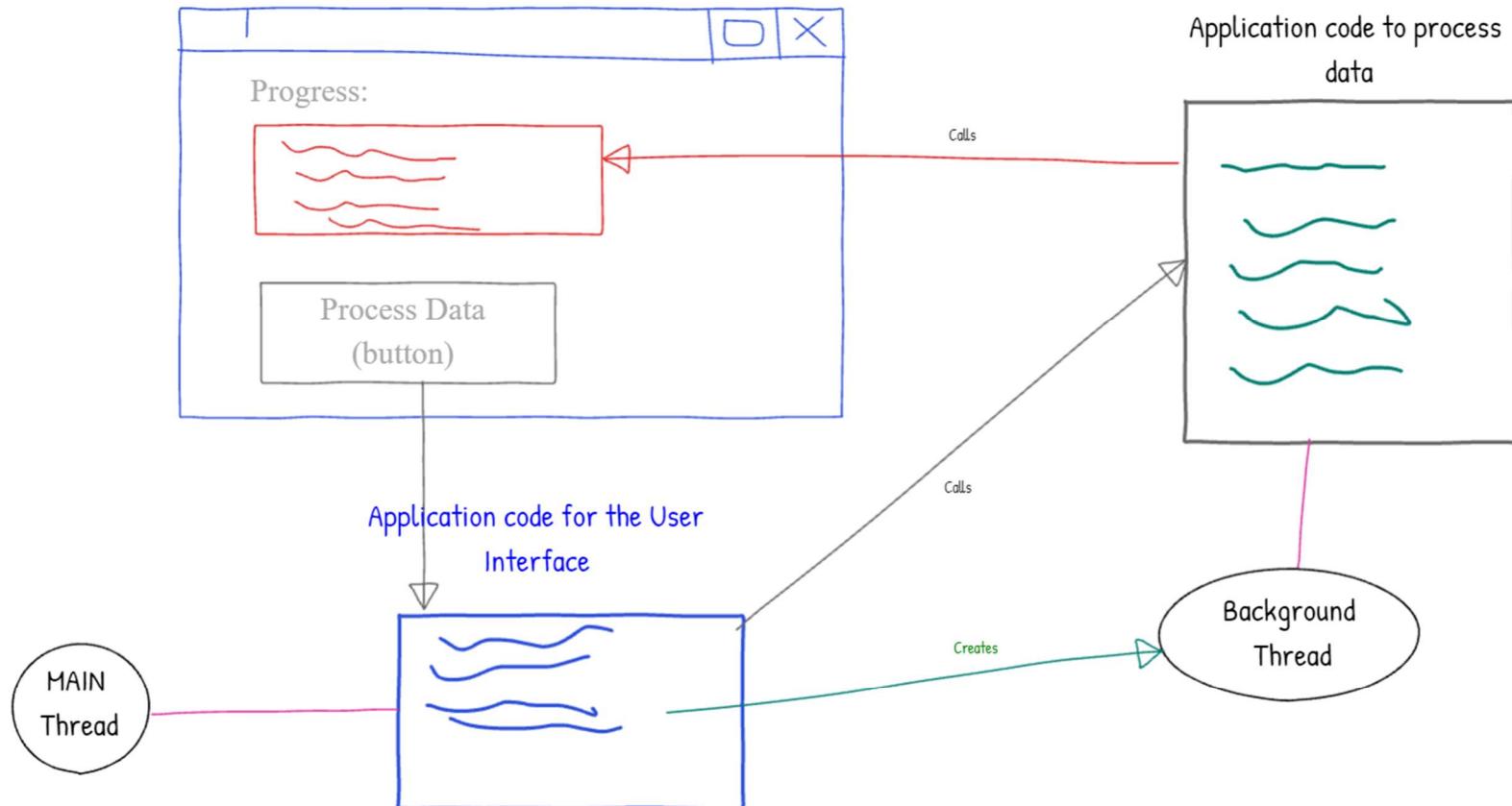
These are execution paths through the application. Can you see they look a bit like threads of cotton?

If the hardware on your device allows both paths to execute simultaneously then it is said to be multi-threaded.



## MODULE 10

Why do we need multi-threading? For this application, if the code to process the data was run on the main thread then the user interface would freeze. By running it on a separate thread both things can happen at the same time.



## MODULE 10

```
class Program
{
    //delegate void MyDelegate(int x, int y, string z);
    static void Main(string[] args)
    {
        //MyDelegate x = AddThem;
        Action<int, int, string> x = AddThem;
        x(3, 4, "5");

        Func<int, int, string, int> y = AddThem2;
        Console.WriteLine(y(5,6,"7"));
        Console.ReadLine();
    }

    static void AddThem(int x, int y, string z)
    {
        Console.WriteLine(x + y + Convert.ToInt32(z));
    }

    static int AddThem2(int x, int y, string z)
    {
        return x + y + Convert.ToInt32(z);
    }
}
```

Action is a data type  
Action is a generic  
Action can point to a method  
that returns a void

Func is a data type  
Func is a generic  
Func can point to a method  
that returns a value

## MODULE 10

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Type a number");
        string x = Console.ReadLine();

        Task T = new Task(p => Console.WriteLine(Convert.ToInt32(p) * Convert.ToInt32(p) * Convert.ToInt32(p)), x);

        T.Start();
        T.Wait();

        Console.ReadLine();
    }
}
```

Lambda

This lambda is an inline method  
it accepts one parameter called p

This is the  
value  
passed to p

## MODULE 10

A simple example of running a task. Notice that what the task is to do is defined by a Lambda expression.

```
static void Main(string[] args)
{
    Console.WriteLine("Type a number");
    string x = Console.ReadLine();

    Task<int> T = new Task<int>(p =>
    {
        DoSomethingThatTakesALongTime();
        return Convert.ToInt32(p) * Convert.ToInt32(p) * Convert.ToInt32(p);
    }, x);

    T.Start();
    T.Wait();

    Console.WriteLine($"The cube of {x} is {T.Result}");

    Console.WriteLine("Ok, we're done. press <enter> to end");
    Console.ReadLine();
}

private static void DoSomethingThatTakesALongTime()
{
    decimal result = 0;
    for (decimal i = 0; i < 300000000M; i++)
    {
        result += i;
    }
}
```

## MODULE 10

Starting a Task with a parameter. In this case the parameter is a list which can hold multiple items.

```
static void Main(string[] args)
{
    string s1 = "The ";
    string s2 = "psychopathic ";
    string s3 = "leader ";

    List<string> strings = new List<string> { s1, s2, s3 };

    Task<string> T = new Task<string>(p =>
    {
        DoSomethingThatTakesALongTime();
        string result = "";
        foreach (var s in (List<string>)p)
        {
            result += s;
        }
        return result;
    }, strings);

    T.Start();
    T.Wait();

    Console.WriteLine($"{T.Result}");

    Console.WriteLine("Ok, we're done. press <enter> to end");
}
```

C# assumes the datatype  
of this parameter is 'Object'

So that means I have to cast (convert)  
it to List<string>

## MODULE 10

### Parallel Invoke

```
6         InitializeComponent();
7     }
8
9     private void Button_Click_1(object sender, RoutedEventArgs e)
10    {
11        Parallel.Invoke(() =>
12        {
13            decimal result = 0;
14            for (decimal i = 0; i < 40000000M; i++)
15            {
16                result += i;
17            }
18            MessageBox.Show(String.Format("40000000! is {0}", result));
19        },
20
21        () =>
22        {
23            decimal result = 0;
24            for (decimal i = 0; i < 20000000M; i++)
25            {
26                result += i;
27            }
28            MessageBox.Show(String.Format("20000000! is {0}", result));
29        }
30    }
31
32    this lambda
33    has 0 paramters );
34
35
```

task 1

task 2

## MODULE 10

### Task ContinueWith

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ConsoleApplication1
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            // Create a task that returns a string.
14            Task<string> firstTask = new Task<string>(() => "Hello");
15
16            // Create the continuation task.
17            // The delegate takes the result of the antecedent task as an argument.
18            Task<string> secondTask = firstTask.ContinueWith((antecedent) => String.Format("{0}, World!", antecedent.Result));
19
20            // Start the antecedent task.
21            firstTask.Start();
22
23            secondTask.Wait();
24            Console.WriteLine(secondTask.Result);
25            Console.WriteLine("Done. Press enter to exit");
26            Console.ReadLine();
27
28        }
29    }
}
```

how many tasks are there? 2

what does the first task return?

where does this get it's value?

what is that? parameter

what is that? Lambda

when does second task start?  
after the first task ends

when does the first task start?

## MODULE 10

If a worker thread tries to access a WPF user control that was created on the main thread, you will get a cross-threading exception. This is solved by either using `Dispatcher.BeginInvoke` or by using the async-await pattern.

The screenshot shows the Visual Studio IDE with the following details:

- Project:** WpfApplication14
- File:** MainWindow.xaml.cs
- Thread:** [4780] Worker Thread
- Stack Frame:** WpfApplication14.MainWindow.NormalState()

```
MainWindow.xaml.cs
1 //Ignore these files/folders
2
3     return filesInPath;
4 }
5
6
7
8
9
10
11
12
13
14     private void CountingState()
15     {
16         btnCount.IsEnabled = false;
17         btnCancel.IsEnabled = true;
18         lblMessage.Content = "Counting in progress....Please wait...";
19     }
20
21     private void NormalState()
22     {
23         btnCount.IsEnabled = true; // Line 103 circled in red
24         btnCancel.IsEnabled = false;
25         lblMessage.Content = "";
26     }
27 }
```

A red arrow points from the text "the line that failed is running on the worker thread" to the circled line 103 in the code.

A blue callout bubble contains the text "solution is the worker thread calls the main thread". A blue arrow points from this bubble to the line 103 in the code.

A red arrow points from the text "main thread owns the button" to the line 103 in the code.

An error dialog box titled "Exception User-Unhandled" is displayed, showing the message: "System.InvalidOperationException: 'The calling thread cannot access this object because a different thread owns it.'".

Buttons in the dialog box include "View Details", "Copy Details", and "Exception Settings".