20. Introduction to Classes

The Object Oriented Toaster …

Once upon a time, in a kingdom not far from here, a king summoned two of his advisors for a test. He showed them both a shiny metal box with two slots in the top, a control knob, and a lever. "What do you think this is?"

One advisor, an Electrical Engineer, answered first. "It is a toaster," he said.

The king asked, "How would you design an embedded computer for it?"

The advisor: "Using a four-bit microcontroller, I would write a simple program that reads the darkness knob and quantifies its position to one of 16 shades of darkness, from snow white to coal black.

The program would use that darkness level as the index to a 16-element table of initial timer values. Then it would turn on the heating elements and start the timer with the initial value selected from the table.

At the end of the time delay, it would turn off the heat and pop up the toast. Come back next week, and I'll show you a working prototype."

The second advisor, a software developer, immediately recognized the danger of such short-sighted thinking. He said, "Toasters don't just turn bread into toast, they are also used to warm frozen waffles.

What you see before you is really a breakfast food cooker. As the subjects of your kingdom become more sophisticated, they will demand more capabilities.

They will need a breakfast food cooker that can also cook sausage, fry bacon, and make scrambled eggs. A toaster that only makes toast will soon be obsolete. If we don't look to the future, we will have to completely redesign the toaster in just a few years."

"With this in mind, we can formulate a more intelligent solution to the problem. First, create a class of breakfast foods. Specialize this class into subclasses: grains, pork, and poultry.

The specialization process should be repeated with grains divided into toast, muffins, pancakes, and waffles; pork divided into sausage, links, and bacon; and poultry divided into scrambled eggs, hard- boiled eggs, poached eggs, fried eggs, and various omelette classes."

"The ham and cheese omelette class is worth special attention because it must inherit characteristics from the pork, dairy, and poultry classes. Thus, we see that the problem cannot be properly solved without multiple inheritance.

At run time, the program must create the proper object and send a message to the object that says, 'Cook yourself.' The semantics of this message depend, of course, on the kind of object, so they have a different meaning to a piece of toast than to scrambled eggs."

"Reviewing the process so far, we see that the analysis phase has revealed that the primary requirement is to cook any kind of breakfast food.

In the design phase, we have discovered some derived requirements.

Specifically, we need an object-oriented language with multiple inheritance. Of course, users don't want the eggs to get cold while the bacon is frying, so concurrent processing is required, too."

"We must not forget the user interface. The lever that lowers the food lacks versatility, and the darkness knob is confusing. Users won't buy the product unless it has a user-friendly, graphical interface.

When the breakfast cooker is plugged in, users should see a cowboy boot on the screen. Users click on it, and the message 'Booting UNIX v.8.3' appears on the screen. (UNIX 8.3 should be out by the time the product gets to the market.) Users can pull down a menu and click on the foods they want to cook."

"Having made the wise decision of specifying the software first in the design phase, all that remains is to pick an adequate hardware platform for the implementation phase. An Intel Pentium with 48MB of memory, a 1.2GB hard disk, and a SVGA monitor should be sufficient.

If you select a multitasking, object oriented language that supports multiple inheritance and has a built-in GUI, writing the program will be a snap."

The king wisely had the software developer beheaded, and they all lived happily ever after.

From Reddit More fun on the link…

Properties and their Generation

What are Properties?

Properties provide the opportunity to protect a **field** in a class by reading and writing to it through that property.

Properties combine aspects of both fields (variables) and methods.

```
        //****Declare Properties ****
        public double num1 { get; set; }
        public double num2 { get; set; }
```

To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a **get** accessor and/or a **set** accessor.

The code block for the get accessor is executed when the property is **read (data out)**; the code block for the set accessor is executed when the property is **assigned a new value (data in)**.

A property without a set accessor is considered read-only. A property without a get accessor is considered write-only. A property that has both accessors is read-write.

Always favour properties over fields for public members. An automatically implemented property looks like a field to the outside world, but you can always add extra behaviour when necessary.

Automatically generate a Short Property layout

Type in **prop** and press Tab two times. This …

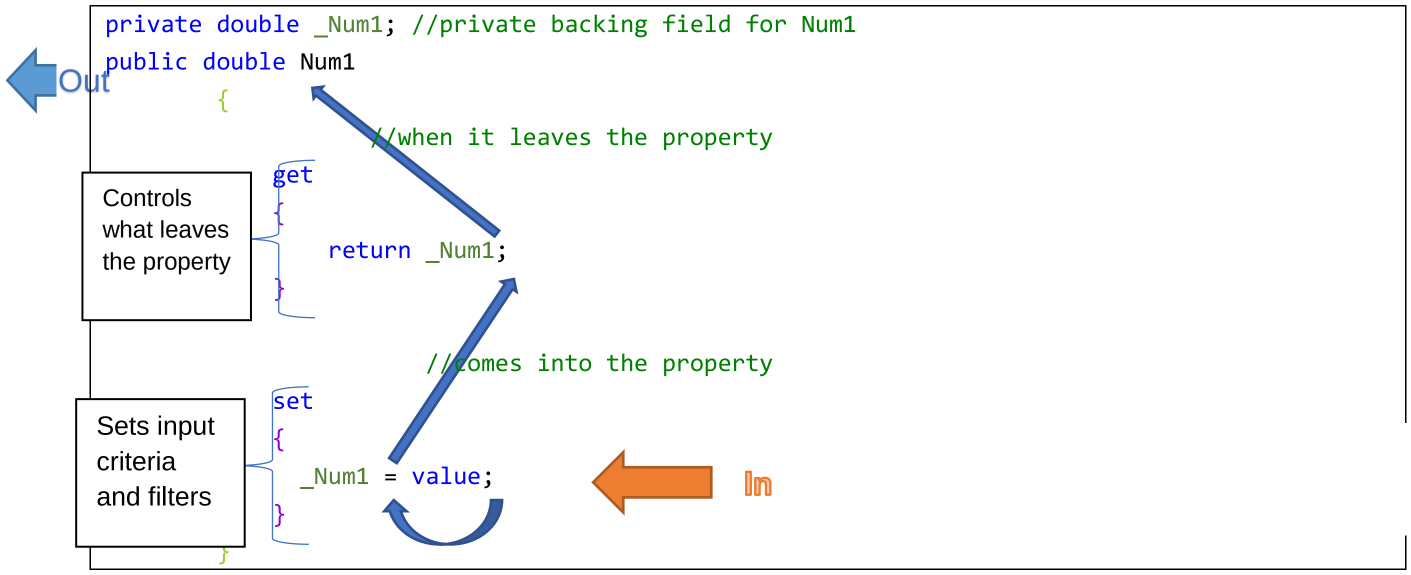
```
public double Num1 { get; set; }
```

… is the same as this …..

Type **Propfull** and press tab this will give full properties including the fields you need that can be automatically changed as needed.

```
private double _Num3;
public double Num3 {
            get { return _Num3; }
            set { _Num3 = value; }
        }
```

Which is the same as this ……


Out
Controls what leaves the property
Sets input criteria and filters
In

```
private double _Num1; //private backing field for Num1
public double Num1
        {
                   //when it leaves the property
            get
            {
                return _Num1;
            }
                       //comes into the property
            set
            {
              _Num1 = value;
            }
        }
```

Unless you are going to use the Get and Set for business rules and business logic then only use **short properties**.

Automatically generate Full Properties with Propfull

Type **Propfull** and press tab this will give full properties including the fields you need that can be automatically changed as needed.

```
private int _MyProperty;
      public int MyProperty
      {
            get { return _ MyProperty;}
            set { _MyProperty = value;}
      }
```

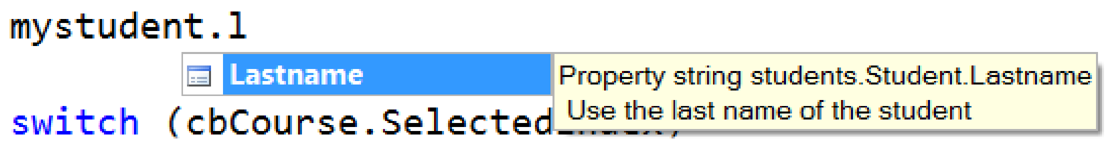
Automatically generate Summary Comments that appear on the tooltip

Press /// three times, and a summary code block is added to your code.

```
/// <summary>
/// Use the last name of the student
/// </summary>
```

If you put it directly above the property then the tooltip will appear on that property

```
        public String Lastname { get; set; }
```

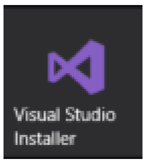


```
mystudent.l
switch (cbCourse.SelectedIndex)
```

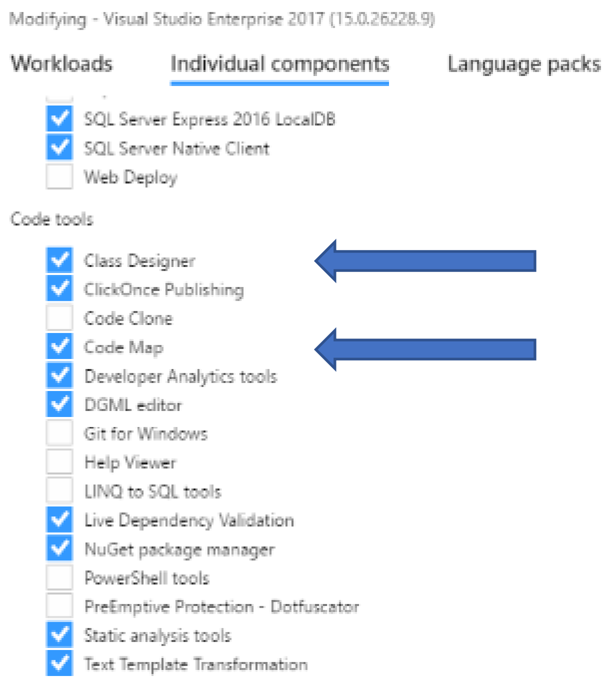
Property string students.Student.Lastname
Use the last name of the student

How to create Class Diagrams and Code Map

Add in the Two Components. Open the Visual Studio Installer



What happened to Class Diagrams?


Modifying - Visual Studio Enterprise 2017 (15.0.26228.9)

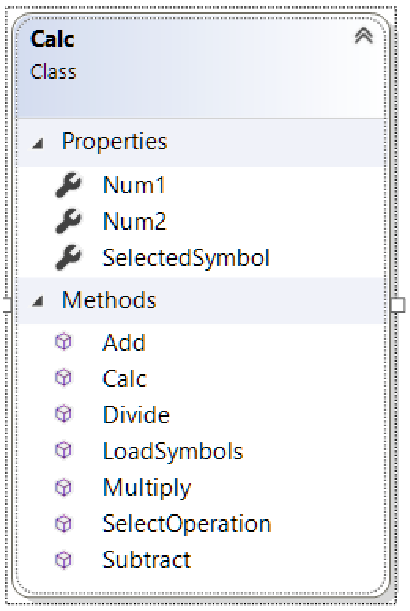
Workloads | Individual components | Language packs

- [✓] SQL Server Express 2016 LocalDB
- [✓] SQL Server Native Client
- [ ] Web Deploy

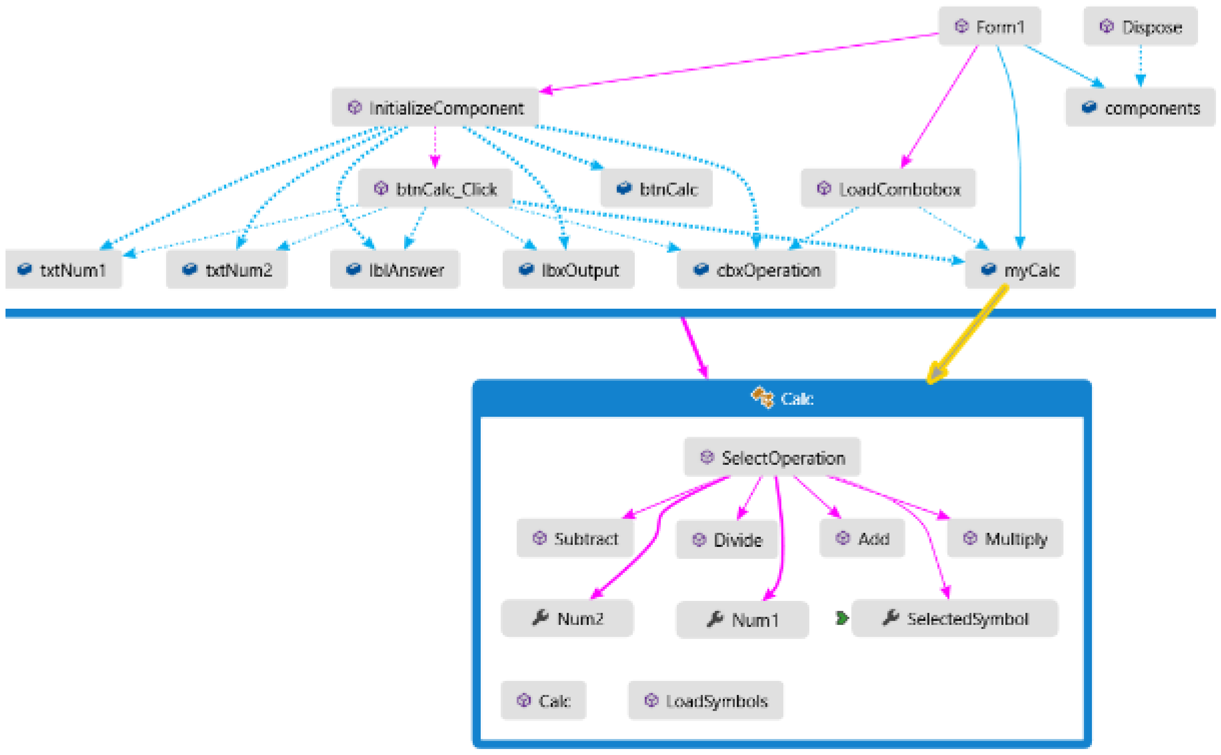
Code tools

- [✓] Class Designer
- [✓] ClickOnce Publishing
- [ ] Code Clone
- [✓] Code Map
- [✓] Developer Analytics tools
- [✓] DGML editor
- [ ] Git for Windows
- [ ] Help Viewer
- [ ] LINQ to SQL tools
- [✓] Live Dependency Validation
- [✓] NuGet package manager
- [ ] PowerShell tools
- [ ] PreEmptive Protection - Dotfuscator
- [✓] Static analysis tools
- [✓] Text Template Transformation

Class Designer



Code Map



New C# Property features in C# 6

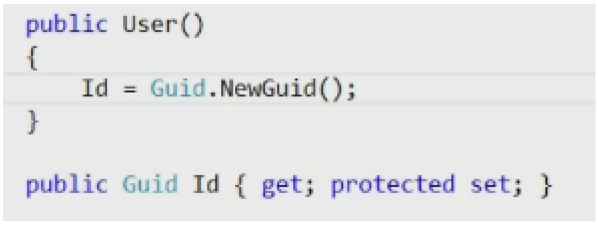
http://www.pluralsight.com/courses/csharp-6-whats-new

Auto Property Initializers.

You can use an auto implemented property and still give it an initial value in one action.

So here we are assigning the value to the Get. Therefore, you don’t need to write a constructor method to pass the data across.

Old way…

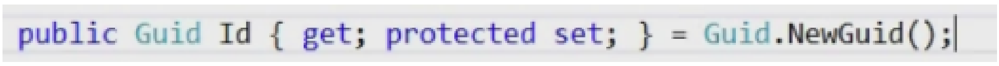


```
public User()
{
    Id = Guid.NewGuid();
}

public Guid Id { get; protected set; }
```

This is instead of using the Constructor methods and then assigning the value to the property.

New way…



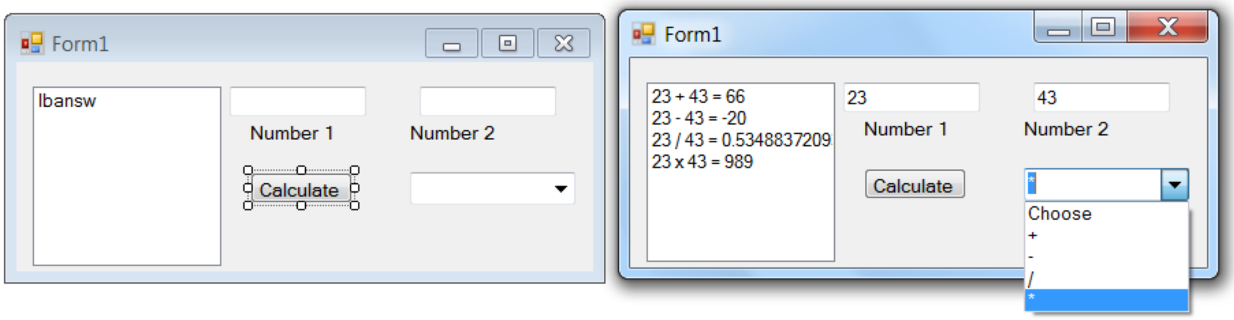
```
public Guid Id { get; protected set; } = Guid.NewGuid();
```

Guid.NewGuid creates a unique globally unique identifier (GUID)

This is a convenient **static** method that you can call to get a new Guid.

Calculator with Class – Intro to classes

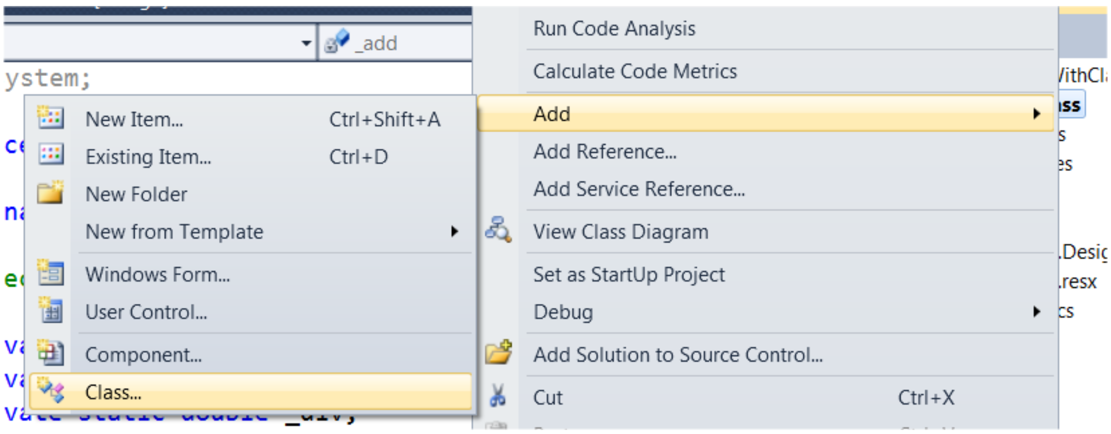
This is an introductory lesson, to show the principles behind OOP. We are going to make another calculator.



Let’s start by making the Class then putting the Form code last.

The Calc class.

To add a class to your project right click on the name of the project and go Add and click on class



Name it **Calc**.

The Standard class layout

The standard layout for a class is as follows. We will cover all these internals later. This lesson just uses **Fields**, **Properties**, and **Methods**

- Fields
- Constructors
- Nested Enums, structs, classes
- Properties
- Methods

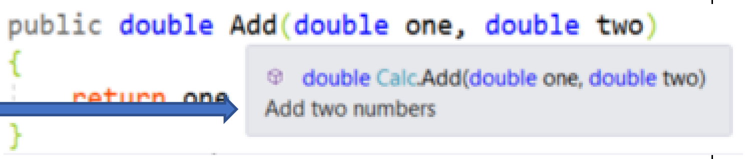
Here is our class below we can see it by going to **View Class Diagram** by right clicking on the class file

```
namespace CalcWithClass
{
public class Calc    {
        //****Declare Fields - None ****
        //****Declare Properties ****
        public double Num1 { get; set; }
        public double Num2 { get; set; }

//show the symbol in the listbox
        public string SelectedSymbol { get; set; }
         //****Set up the constuctors****
        //(no constructors needed)

        //****Create the methods****

         /// <summary>
        /// Add two numbers
        /// </summary>
        public double Add(double one, double two)
        {
            return one + two;
        }
        /// <summary>
        /// Subtract two numbers
        /// </summary>
        public double Subtract(double one, double two)
        {
            return one - two;
        }
        /// <summary>
        /// Divide two numbers
        /// </summary>
        public double Divide(double one, double two)
        {
            return one / two;
        }
        /// <summary>
```



```
public double Add(double one, double two)
{
    return one
}
```

double Calc.Add(double one, double two)
Add two numbers

```
        /// Multiply two numbers
        /// </summary>
        public double Multiply(double one, double two)
        {
            return one * two;
        }


        /// <summary>
        /// Load the symbols to the ComboBox
        /// </summary>
        public string[] LoadSymbols()
        {
            //make an array of symbols
String[] ComboBoxItems = new[] { "Choose an operation", "*", "-", "+", "/" };
            return ComboBoxItems;
        }
        /// <summary>
        /// Input the Symbol and select the calculation
        /// </summary>
        public double SelectOperation()
        {
            double Answer = 0;
            switch (SelectedSymbol)
            {
                case "*":
                    Answer = Multiply(Num1, Num2);
                    break;
                case "-":
                    Answer = Subtract(Num1, Num2);
                    break;
                case "+":
                    Answer = Add(Num1, Num2);
                    break;
                case "/":
                    Answer = Divide(Num1, Num2);
                    break;
            }
            return Answer;
        }
```

Form Details

To join our calc class to our form we have to declare it and create a new instance of it

```
        private calc mycalc = new calc();
```

This means that everything in the **calc** class that is public can be access by myCalc

```
public partial class Form1 : Form
    {
        //create a new instance of the Calc class
        private Calc myCalc = new Calc();

        public Form1()
        {
            InitializeComponent();
            LoadCombobox();
        }

        private void LoadCombobox()
        {
            //pass the array across to the combobox
            cbxOperation.Items.AddRange(myCalc.LoadSymbols());
            //start at the first item
            cbxOperation.SelectedIndex = 0;
        }
        //the button click,
        private void btnCalc_Click(object sender, EventArgs e)
        {
            double result;
            //Check for numbers coming in
            if (double.TryParse(txtNum1.Text, out result))
            {
                myCalc.Num1 = result;
            }

            if (double.TryParse(txtNum2.Text, out result))
            {
                myCalc.Num2 = result;
            }
            //check selected operation
            myCalc.SelectedSymbol = cbxOperation.SelectedItem.ToString();

            //output answer
            lblAnswer.Text = Convert.ToString(myCalc.SelectOperation());
lbxOutput.Items.Add(myCalc.Num1 + " " + myCalc.SelectedSymbol + " " + myCalc.Num2 + " = " +
lblAnswer.Text);
        }
    }
```