

**C07215/C07515**

# **Advanced Web Technologies**

## **Lab 1**

Dr Stephan Reiff-Marganiec  
Department of Computer Science

September 2013



### A few words about Labs in Advanced Web Technologies

Before we start with the content, we would like to remind you that in the labs in this module you are learning new material – that is you are learning practical skills that are not covered in the lectures. Especially the first few labs on C# are not about material that is taught in class, but they are rather practical skills that you need to build applications of the type discussed in the lectures.

The approach to working on this material is to read the instructions and then start the coding! Also, if you are a campus-based student we are not expecting that you always finish all the lab material in the allocated lab-time. We rather expect that you spend some of your own time preparing before the lab session and finishing things afterwards.

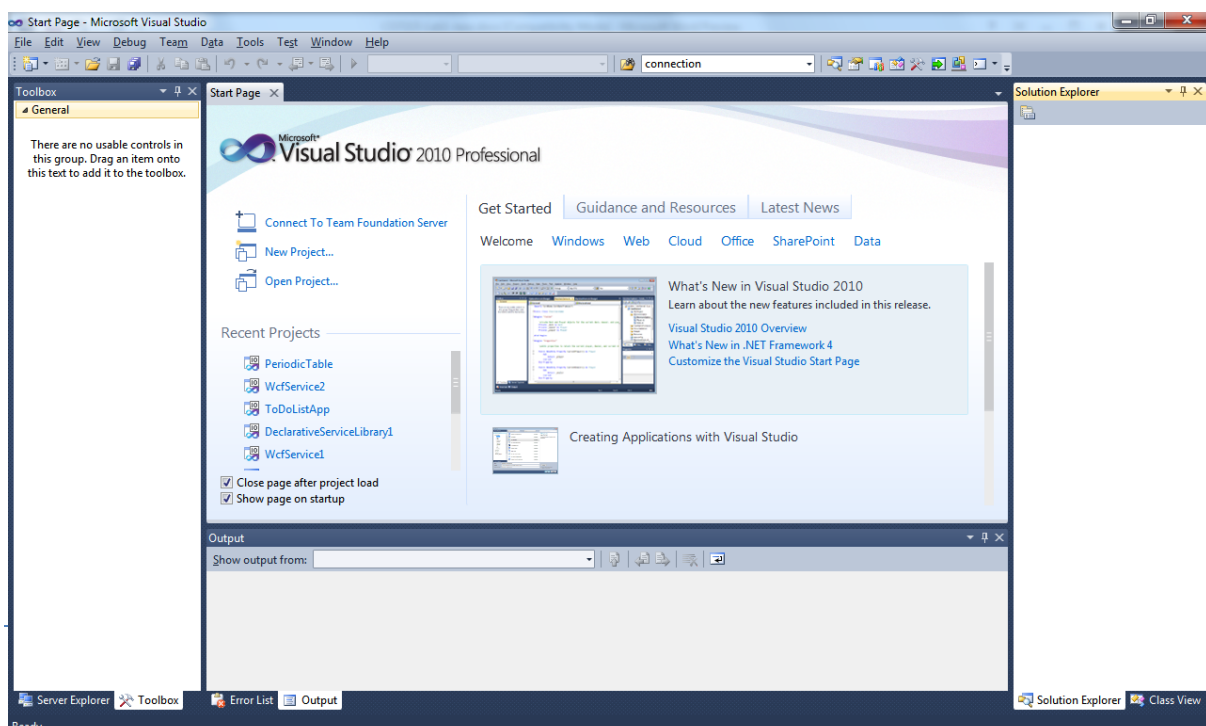
### Lab 1: Visual studio 2010 IDE and C# introduction

In this lab, you will learn to use Visual Studio 2010 and how to write simple applications C#. There is an assumption that you are already familiar with object-oriented programming – if not you might need to work harder. Specifically, you will learn to

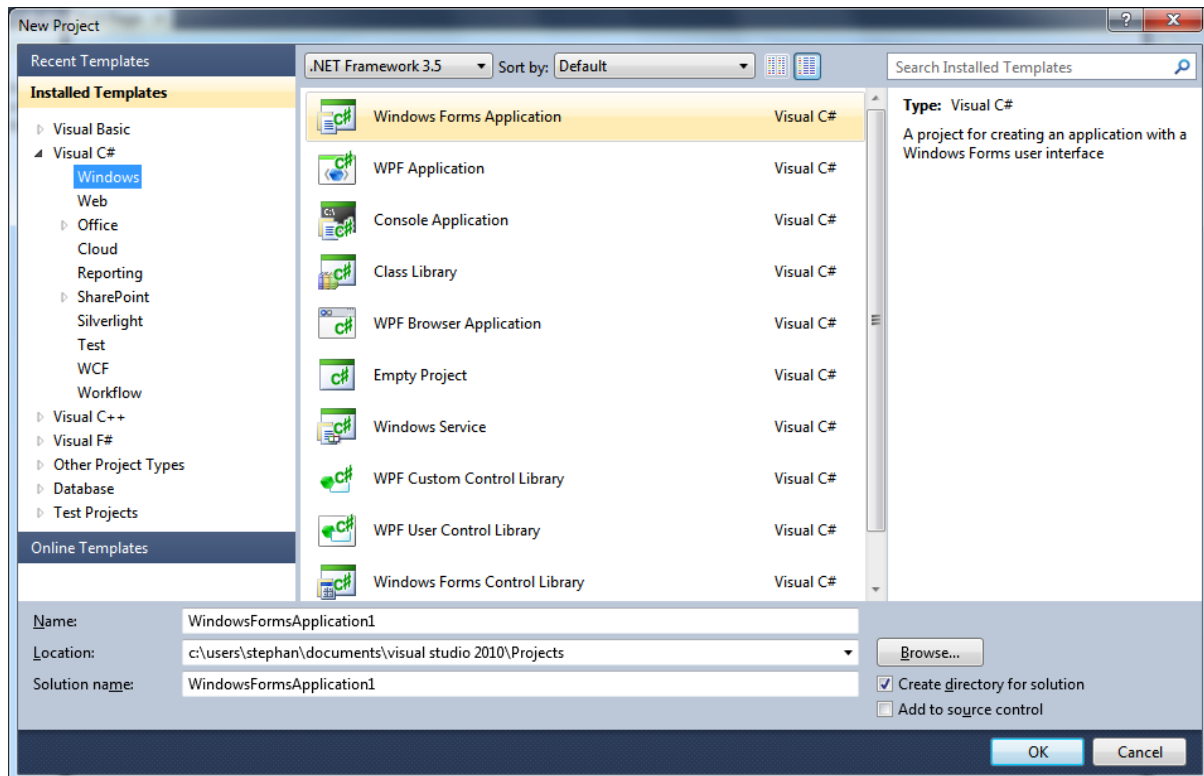
- build a new project and solution and understand C# Console application structure
- understand namespaces, class, method, constructor and basic data types
- use basic control structures
- use exception handling
- use properties and indexers
- understand the concept of Interface and Structs
- use Overload for defining the method with the same name but different interfaces
- implement abstract methods and an interface's method
- understand the concepts of Inheritance and Polymorphism

### Visual studio 2010

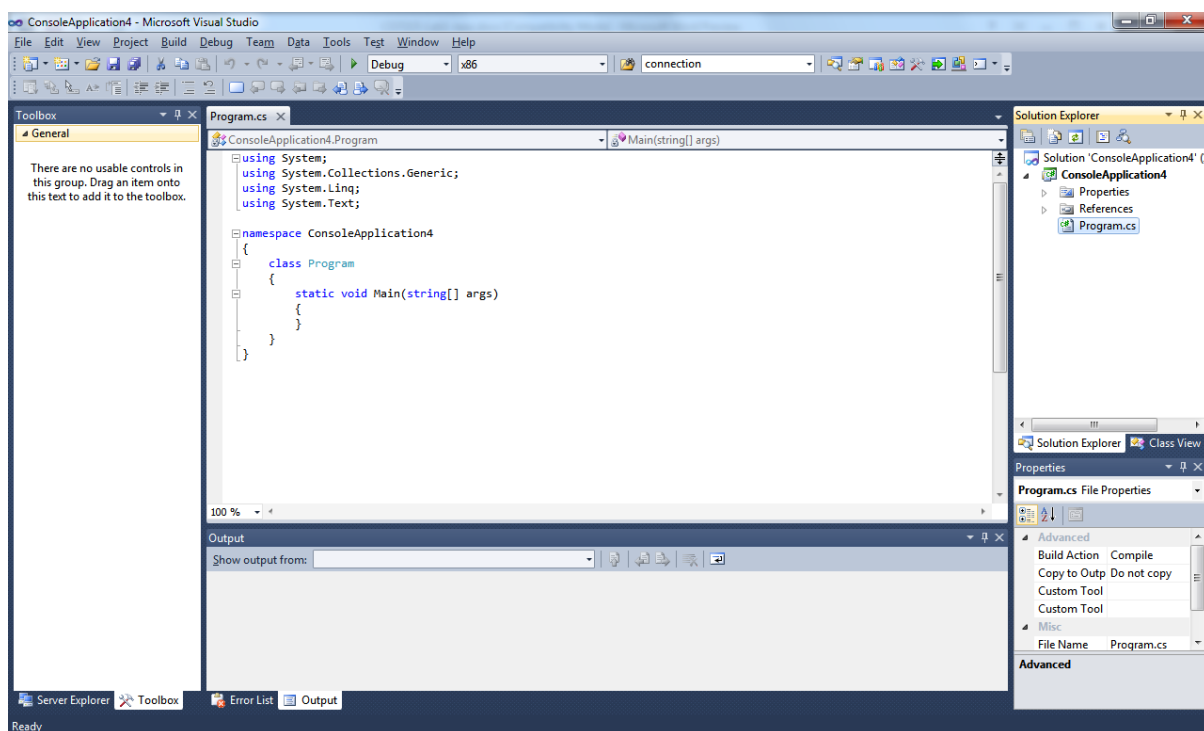
Start Visual studio 2010 from your start menu. You will see the start page.



Create a new project, go to File -> New -> Project, and then you will see the dialog window.



You can select different Templates for different project types. For now click Visual C# and select Console Application. Provide a project Name and Solution Name, but keep the Location path. Click ok and see the new project window appearing.



**Practice 1:** Let us enter the code below inside the Main method:

```
String Name;  
Console.WriteLine("Your name is : ");  
Name=Console.ReadLine();  
Console.WriteLine("Hello "+Name);  
Console.WriteLine();  
Console.WriteLine("Press any key to exit");  
Console.ReadLine();
```

Click the green arrow button to run the program.

### C# Console applications

The C# language can be used to create applications that display on a black window referred to as the DOS prompt or DOS window (or sometimes terminal). In this lab we will create such command line applications, and we have already created one in the last exercise.

The C# program structure includes four parts. The first part is using “References”. References are essentially extra libraries which will be used in the program, for example:

```
using System;  
using System.Collections.Generic;  
using System.Text;
```

The second part is declaring the “namespace” of the programme. Namespace are used in .NET to organize class libraries into a hierarchical structure. One reason to do this is to help organize classes in a meaningful way that is understood by consumers of the class library. Another reason to use namespaces is to reduce naming conflicts of the classes between different organizations.

The third part is declaring the “class”. Classes are the basic ingredients of OO programming. In C#, the classes are declared using the *class* keyword followed by the class name and brackets surrounding the body of the class.

The fourth part is the method coding. Every C# program must have a *Main* method for running the application. This is specified as `static void Main(string[] args) {...}`. Other methods are declared using access modifier [output type] name ([[input type input name], ...]]. For instance,

```
private string Person(string name, int age)  
{  
    string nameage;  
    nameage = name + age.ToString();  
    return nameage;  
}
```

One special method is called *Constructor*, a constructor is obtained when the method's name is the same as the class name and no output type is given. The purpose of the constructor method is to initialize the class for supporting data values for other method to use. The constructor method is optional.

The *access modifier* has five types as shown in the table:

Access Modifier	Visibility
public	Accessible from anywhere
protected	Accessible from this class or any class derived from this class
internal	Accessible with current program only
protected internal	Accessible within current program or any class derived from this class
private (default)	Accessible only within current class

**Practice 2:** Now let us practice a bit. Start a new project in this solution naming it `Add1to10`. Then define a class `Adding` which has a private method called `addnumber ( )`. The method has no input parameter and is meant to calculate and print the sum of all numbers from 1 to 10. You will need a `main` method and a call `do addnumber` to make this work, of course.

### Data Types

From the above examples, we can see that we need to define the data type for output, input attributes and the variables. There are two important kinds of parameters: *Value types* and *Reference types*. Each variable needs a type as C# is a strongly typed language. Table 2 lists the primitive data types

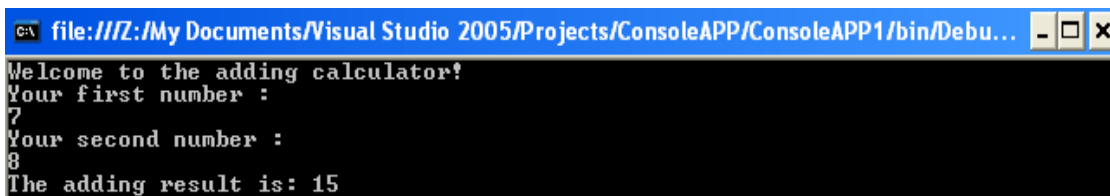
Data Type	Size in Bytes	Description
sbyte	1	Signed byte
byte	1	Unsigned byte
short	2	Signed short
ushort	2	Unsigned short
Int	4	Signed integer
uint	4	Unsigned integer
long	8	Signed long integer
ulong	8	Unsigned long integer
float	4	Floating point
double	8	Double-precision floating point
decimal	8	96-bit signed number
string	n/a	Unicode string
Char	2	Unicode character
bool	n/a	True or false

**Practice 3:** We will now get to a stage with a bit less handholding. Start a new project, to write a calculator application. The console output should be as follows (the welcome sentence is defined in the constructor method):



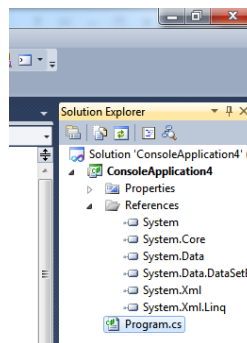
```
C:\ file:///Z:/My Documents/Visual Studio 2005/Projects/ConsoleAPP/ConsoleAPP1/bin/Debu...
Welcome to the adding calculator!
Your first number :
```

Enter the number you like, for instance 7, and then the second number (e.g. 8). Finally print the resulting sum.



```
C:\ file:///Z:/My Documents/Visual Studio 2005/Projects/ConsoleAPP/ConsoleAPP1/bin/Debu...
Welcome to the adding calculator!
Your first number :
7
Your second number :
8
The adding result is: 15
```

Reference types are instance of classes. Reference types are allocated on the heap. In C#, all classes are derived from the .NET framework class `Object` within the `System` namespace. From the solution explorer view you can find out the used references for current project. All available reference types are derived from these references.



### C# Control Structures: *if* and *switch* case statements

The *if* statement executes a series of statements if a test Boolean expression evaluates to true. The test expression to evaluate must be Boolean. The *if-else* statement adds a path for the false evaluation of the Boolean expression. For example,

Example 1:

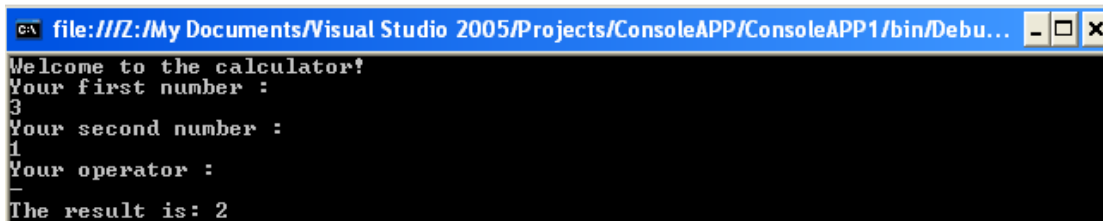
```
if (operator1.Equals("+"))
{
    sum = double.Parse(firstNumber) + double.Parse(secondNumber);
}
if (operator1.Equals("-"))
{
    sum = double.Parse(firstNumber) - double.Parse(secondNumber);
}
```

Example 2:

```
if (operator1.Equals("+"))
{
    sum = double.Parse(firstNumber) + double.Parse(secondNumber);
}
else ()
{
    sum = double.Parse(firstNumber) - double.Parse(secondNumber);
}
```

Do you see the differences between above two examples?

**Practice 4:** Now we program a calculator which takes two numbers and an operator (+, -, \*, /) as input, then give the correct answer as output as shown in the following figure. Additionally, if the input is empty, then an error message should appear.



```
file:///Z:/My Documents/Visual Studio 2005/Projects/ConsoleAPP/ConsoleAPP1/bin/Debu...
Welcome to the calculator!
Your first number :
3
Your second number :
1
Your operator :
-
The result is: 2
```

The *switch* statement chooses flow of control based on the evaluation of a numeric or string comparison. The *switch* statement does not allow control to fall through to the next case unless the case statement is followed immediately by another case statement. For example,

```
switch (number)
{
    case "1":
        n = 1;
        break;
    case "2":
        n = 2;
```

```
        break;
    default:
n = 3;
        break;
}
```

**Practice 5:** Rewrite the calculator from Practice 4 using the switch case statement.

### C# Control Structures: for and while loop statements

The *for* statement is used to loop through a series of statements until a test Boolean expression evaluated at the beginning of the loop is false. For example,

```
for (int i = 0; i < 4; i++)
{
    Console.WriteLine(i);
}
```

**Practice 6:** allow the previous calculator to ask for 4 sets of numbers and operators.

The *while/do while* statement is used to loop through a series of statements until a test Boolean expression evaluated at the beginning/end of the loop is false. For example,

Example 1:

```
int i=0;
while (i<4)
{
    Console.WriteLine(i);
    i++;
}
```

Example 2:

```
int i=0;
do
{
    Console.WriteLine(i);
    i++;
}
while (i < 4);
```

Do you find the differences?

**Practice 7:** allow our calculator continuing to operate until the user inputs "x" to exit.

### C# Control Structures: the break and continue statements

The *break* statement exits the loop in a *for*, *while*, or *do while* statement regardless of the value of the test Boolean expression. The *continue* statement will pass flow of control immediately to the start of a loop when encountered.



**Practice 8:** try to use *break* and *continue* statements in your last calculator.

### Using Properties

In C# *properties* are method calls that look like direct access to member data. Let us start a new project of EmployeeStore and enter the following code:

```
class EmployeeStore
{
    static void Main(string[] args)
    {
        try
        {
            // Create a new employee
            Employee employee = new Employee();

            // Set some properties
            employee.FirstName = "Timothy";
            employee.MiddleName = "Arthur";
            employee.LastName = "Tucker";
            employee.SSN = "555-55-5555";

            // Show the results on screen
            string name = employee.FirstName + " "
                + employee.MiddleName +
                " " + employee.LastName;
            string ssn = employee.SSN;

            Console.WriteLine("Name: {0}, SSN: {1}", name, ssn);

            Console.ReadLine();
        }
        catch (Exception exception)
        {
            // Display any errors on screen
            Console.WriteLine(exception.Message);
        }
    }
}

class Employee
{
    private string m_firstName;
    private string m_middleName;
    private string m_lastName;
    private string m_SSN;

    // FirstName property
    public string FirstName
    {
        get { return m_firstName; }
        set { m_firstName = value; }
    }
}
```

```

    }

    // MiddleName property
    public string MiddleName
    {
        get { return m_middleName; }
        set { m_middleName = value; }
    }

    // LastName property
    public string LastName
    {
        get { return m_lastName; }
        set { m_lastName = value; }
    }

    // SSN property
    public string SSN
    {
        get { return m_SSN; }
        set { m_SSN = value; }
    }
}

```

We take the SSN property as example. The *public* keyword indicates its visibility. Next, this property works with string data as indicated by the *string* keyword. Finally, the name of the property is SSN.

The get accessor method is relatively simple. It just returns the value of the private data member *m\_SSN*. In the program, you can see the SSN property is accessed using syntax usually reserved for accessing member data: `string ssn = employee.SSN;`. The set property is implemented as `employee.SSN = "555-55-5555".`

**Practice 9:** Add the name as a property, too.

### Using Indexers

The need to create and manipulate lists is a common programming task. Let's extend our employee example from the last section. Let's say you need to display a list of employees. The most logical thing to do would be to create a new *Employees* class, which contains all of the individual *Employee* instances. You would then iterate through all of the employees displaying each one until there are no further employees. One way to solve this would be to create a property that returns the number of employees and a method that returns a given employee given its position in the list, one possible solution is:

```

for ( i = 0; i < employees.Length; i++ )
{
    Employee employee = employees.GetEmployee( i );
    Console.WriteLine( employee.LastName );
}

```

However, it would be more intuitive if we could just treat the list of employees as an array containing employee objects. Here is what that may look like:

```
for ( i = 0; i < employees.Length; i++ )
{
    Console.WriteLine( employees[i].LastName );
}
```

**Practice 10:** download the Indexers.cs file from the lab resources in Blackboard and try to understand the indexers more deeply.

### Exception Handling

Exceptions are unforeseen errors that happen in your programs. Most of the time, you can, and should, detect and handle program errors in your code. For example, validating user input, checking for null objects, and verifying the values returned from methods are what you expect, are all examples of good standard error handling that you should be doing all the time. However, there are times when you don't know if an error will occur – especially in web or distributed systems. For example, you can't predict when you'll receive a file I/O error, run out of system memory, or encounter a database error. These things are generally unlikely, but they could still happen and you want to be able to deal with them when they do occur. This is where exception handling comes in. When exceptions occur, they are said to be "thrown". What is actually thrown is an object that is derived from the *System.Exception* class. In the next section, I'll be explaining how thrown exceptions are handled with *try/catch* blocks.

Identifying the exceptions you'll need to handle depends on the routine you're writing. For example, if the routine opened a file with the *System.IO.File.OpenRead()* method, it could throw any of the following exceptions:

- *SecurityException*
- *ArgumentException*
- *ArgumentNullException*
- *PathTooLongException*
- *DirectoryNotFoundException*
- *UnauthorizedAccessException*
- *FileNotFoundException*
- *NotSupportedException*

It's easy to find out what exceptions a method can raise by looking in the .NET Frameworks SDK Documentation. Just go to the Reference/Class Library section and look in the Namespace/Class/Method documentation for the methods you use. Here is a simple exception example:

```
using System;
using System.IO;
```

```
class tryCatchDemo
{
    static void Main(string[] args)
    {
        try
        {
            File.OpenRead("NonExistentFile");
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```

Although the code has only got a single *catch* block, all exceptions will be caught there because the type is of the base exception type "Exception". In exception handling, more specific exceptions will be caught before their more general parent exceptions. For example, the following snippet shows how to place multiple catch blocks:

```
catch(FileNotFoundException fnfex)
{
    Console.WriteLine(fnfex.ToString());
}
catch(Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

An exception can leave your program in an inconsistent state by not releasing resources or doing some other type of cleanup. A *catch* block is a good place to figure out what may have went wrong and tries to recover, however it can't account for all scenarios. Sometimes you need to perform clean up actions whether or not your program succeeds. These situations are good candidates for using a *finally* block. Below is an example of using *finally*.

```
using System;
using System.IO;

class FinallyDemo
{
    static void Main(string[] args)
    {
        FileStream outputStream = null;
        FileStream inputStream = null;

        try
        {
            outputStream = File.OpenWrite("DestinationFile.txt");
```

```

        inStream = File.OpenRead("BogusInputFile.txt");
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    finally
    {
        if (outStream != null)
        {
            outStream.Close();
            Console.WriteLine("outStream closed.");
        }
        if (inStream != null)
        {
            inStream.Close();
            Console.WriteLine("inStream closed.");
        }
    }
}

```

You can also throw exceptions to indicate errors that occur in your programs by using *throw* keywords. To throw an exception, you create a new instance of a `System.Exception` class that indicates the type of exception encountered.

```
throw new System.Exception();
```

**Practice 11:** Add a try and catch statement to the last version of the calculator code to indicate that there is an error when there are no input numbers or operators.

### Structs

A *struct* is a simple user-defined type, a lightweight alternative to a class. *Structs* are similar to classes in that they may contain constructors, properties, methods, fields and indexers and so on. However, there are significant differences between classes and *structs*. For instance, *structs* do not support inheritance (which we will see later in today's lab). More importantly, classes are passed by reference while *structs* are passed by value – this has implications on the memory use when you keep these structures and furthermore has an impact on changes you make to them.

```

public struct Address
{
    private string street;

    public Address(string str)
    {
        street = str;
    }
    public string s
    {

```

```

        get
        {
            return street;
        }
        set
        {
            street = value;
        }
    }
}

```

You cannot initialize an instance field in a struct, for example:

```
private string street= "London Road";
```

**Practice 12:** add an address attribute to the employee class from last week. The Address struct should contain a street, postcode and country attributes. Adjust your earlier code to add an employee into the store with his specific address details.

**Practice 13:** think about what happens if you pass a struct as parameter to a method and then change it inside the method. What has happened to the struct in the calling method? You might want to implement a simple example: A class that initializes a struct, calls a method that updates a field in the struct and then (in the main class) prints the values of the struct.

## Interface

An *interface* is a contract that guarantees to a client how a class or struct will behave. An interface offers an alternative to an abstract class for creating contracts among classes and their clients. These contracts are made using the *interface* keyword, which declares a reference type that encapsulates the contract.

```

interface QueryEmployee
{
    void Query();
}

```

This example shows that there is a QueryEmployee interface which promises that any class implementing the interface should have a Query() method. C# allows for a class to have more than one interface inheritances. The inheritance is declared by:

```
class Employee: queryEmployee
```

**Practice 14:** Let us implement an UpdateEmployee interface which has update, delete, and insert behaviours. Then, we declare the employee class to commit to this interface as well as the above QueryEmployee interface. You should implement the respective behaviours inside the Employee class.

## Overload

Often the programmer wants to have more than one function with the same name. The most common example of this is to have more than one constructor. For example

```
class Employee
{
    private string m_firstName;
    private string m_middleName;
    private string m_lastName;
    private double m_HourlyRate;
    private double m_HourWorked;
    private double year_salary;
    private int age;
    private string m_SSN;

    public Employee()
    {
        int default_age = 18;
        age = default_age;
    }
    public Employee(int m_age)
    {
        age = m_age;
    }
}
```

We can see that there are two different constructors. One constructor does not have input attributes and the other one has the `m_age` input attribute. We also can declare two different non-constructor methods with the same name, thus effectively overriding one by the other and allowing methods with different input arguments.

**Practice 15:** Based on the `Employee` class, add two monthlySalary methods to the class. One method uses `year_salary/12` and the other one use `m_HourlyRate * m_HourWorked`. Finally, we add two people as employee. One person has an annual salary of 20000 and the other has the hourly salary (hourly rate is 10 and there are 140 hour a month). The system should enable to output the name, age, SSN and the monthly salaries of these two people.

## Class Inheritance: Override

Override is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event. For example, the `Square` class must provide an overridden implementation of `Area` because `Area` is inherited from the abstract `ShapesClass`:

```
abstract class ShapesClass
{
    abstract public int Area();
}

class Square : ShapesClass
{
}
```

```

        int side = 0;

        public Square(int n)
        {
            side = n;
        }
        // Area method is required to avoid
        // a compile-time error.
        public override int Area()
        {
            return side * side;
        }

        static void Main()
        {
            Square sq = new Square(12);
            Console.WriteLine("Area of the square = {0}", sq.Area());
        }
    }
    // Output: Area of the square = 144

```

**Practice 16:** Define an interface of an employee store as below. The `Query()` method from the employee class should be overridden to allow to query for an employee's name. The `Input()` method should be overridden to allow adding a new employee to the store.

```

interface queryEmployee
{
    void Query();
    void Input();
}

```

### Class Inheritance: Polymorphism

Inheritance and polymorphism are the two characteristics that make object-oriented programming languages so powerful. Inheritance means you can create a new type of object B that inherits all of the characteristics of an existing object A. Polymorphism means that this new object B can choose to inherit some characteristics and supply its own implementation for others. For example, you have seen examples that use the Employee class. An employee in our case has a first name, middle name, last name, and SSN. Now we want to add in wage information. There are two types of wages. One is based on an annual salary and the other one is based on an hourly wage. In this case, we can use inheritance to obtain a solution. Inheritance lets you create two new types of the employees (yearly and hourly) that inherit all of the characteristics of the employee class. Here are the declarations of the new classes.

```

class YearlyEmployee : Employee
{
}

class HourlyEmployee : Employee
{
}

```



```
}
```

When we want to define an instance of Employee, we need to define the specific class of the Employee: YearlyEmployee or Hourly Employee like this

```
Employee employee = new YearlyEmployee(20000);

employee.FirstName = "Timothy";
employee.MiddleName = "Arthur";
employee.LastName = "Tucker";
employee.SSN = "555-55-5555";
```

**Practice 17:** Let us add YearlyEmployee and HourlyEmployee classes into the project of the Employee store. Add employees to the store, for example one who has an annual salary of 20000 and one who is paid 10 per hour. Create a virtual method called getMonthlySalary inside class Employee as shown below:

```
public virtual double getMonthlySalary(){
    Console.WriteLine("getMonthlySalary() must be overridden");
    return 0;
}
```

To make use of polymorphism, override this method within class YearlyEmployee and HourlyEmployee respectively so that it will be invoked on each of the derived classes, not the base Employee class. Assume they both worked 140 hours a month -- let the system show their names, SSN and monthly income.