

Object-oriented programming and basic .NET concepts

Subjects:

- Object-oriented programming
- Simple C# program
- Compilation and running
- .NET settings
- Syntax explanation
- Variations on the Main() Method
- Processing of command-line arguments
- Using of Microsoft Development Network

Introduction

To reduce the number of issues you deal with, you work at a certain level of detail. In object-oriented (OO) computerese, the level of detail at which you're working is the **level of abstraction**.

Computer scientists have invented **object orientation** and numerous other concepts that **reduce the level of complexity** at which programmers have to work. Using powerful **abstractions makes the job simpler and far less error-prone** than it used to be. In a sense, that's what the past half-century or so of **computing progress has been about: managing ever more complex concepts and structures with ever fewer errors.**

Basic concepts of programming are: **simplicity, clarity and generality**. These concepts are the true way for success in complex projects.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data-fields and methods – and their interactions to design applications and computer programs.

OOP has roots that can be traced to the 1960s. As hardware and software became increasingly complex, quality was often compromised. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The methodology focuses on data rather than processes, with programs composed of self-sufficient modules (objects) each containing all the information needed to manipulate its own data structure.

An **object-oriented program** may thus be viewed as a collection of cooperating objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "operators") on these objects are closely associated with the object.

Procedural programming versus OOP

Procedural approach has these problems:

- ◆ **It's too complex. You don't want the details.** If you can't define the objects and pull them from the morass of details to deal with separately, you must deal with all the complexities of the problem at the same time.
- ◆ **It isn't flexible.**
- ◆ **It isn't reusable.** Having solved a problem once, you want to be able to reuse the solution in other places within my program. If you're lucky, you may be able to reuse it in future programs as well.

C# features

These C# features are necessary for writing object-oriented programs:

- ◆ **Controlled access:** C# controls the way in which class members can be accessed. C# keywords enable you to declare some members wide open to the public whereas internal members are protected from view and some secrets are kept private.
- ◆ **Specialization:** C# supports specialization through a mechanism known as *class inheritance*. One class inherits the members of another class. For example, you can create a `Car` class as a particular type of `Vehicle`.
- ◆ **Polymorphism:** This feature enables an object to perform an operation the way it **wants to**. The `Rocket` type of `Vehicle` may implement the `Start` operation much differently from the way the `Car` type of `Vehicle` does. But all `Vehicles` have a `Start` operation, and you can rely on that.
- ◆ **Indirection.** Objects frequently use the services of other objects — by calling their **public methods**. But classes can “know too much” about the classes they use. The two classes are then said to be “too tightly coupled,” which makes the using class too dependent on the used class. The design is too brittle — liable to break if you make changes. But change is inevitable in software, so you should find more *indirect ways to connect* the two classes. That’s where the C# interface construct comes in.

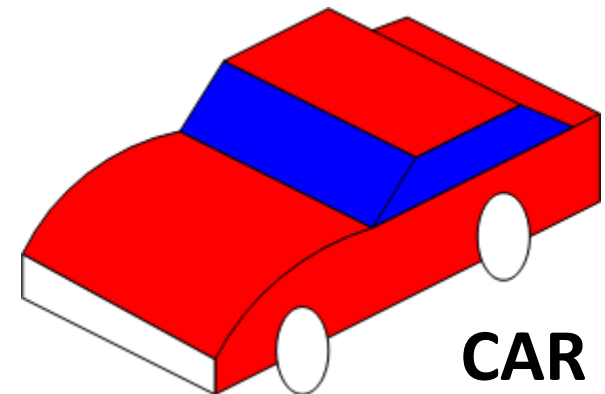
Fundamental concepts and features of OOP

Class. Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something.

Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the **problem domain**, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called **members**.

Object. A pattern (exemplar) of a class. The class of Car defines all possible cars by listing the characteristics and behaviors they can have; the object **FordFocus** is one particular car, with particular versions of the characteristics. A **FordFocus** is model; **FordFocus** has silver body color.

Class Car would consist of traits shared by all cars, such as model and body color (characteristics), and the ability to run and to be driven (behaviors).



Fundamental concepts and features of OOP

Instance. One can have an instance of a class or a particular object. **The instance is the actual object created at runtime.** In programmer jargon, the **FordFocus** object is an instance of the **Car** class. The set of values of the attributes of a particular object is called its **state**. The object consists of state and the behavior that's defined in the object's class.

Method. An object's abilities. In language, methods (sometimes referred to as "**functions**") are verbs. Ford Focus is a car that can be driven. So **drive()** is one of FordFocus methods. She may have other methods as well, for example **on()** or **off()** or **accelerate()** or **decelerate()**. Within the program, using a method usually affects only one particular object.

Message passing. "The process by which an object sends data to another object or asks the other object to invoke a method." Also known to some programming languages as interfacing. For example, the object called **John** may tell the **FordFocus** object to accelerate by passing a "accelerate" message which invokes "accelerate" method of **FordFocus**.



FordFocus

Fundamental concepts and features of OOP

Inheritance. "Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own. Many models of cars are prototypes of previous models (model series).

Multiple inheritance is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other. For example, independent classes could define **Computers** and **MobilePhone**, and a **Communicator** object could be created from these two which inherits all the (multiple) behavior of computers and mobile phones. This is not always supported, as it can be hard both to implement and to use well.

Abstraction. Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

Abstraction is also achieved through composition. For example, a class **Car** would be made up of an **Engine**, **Gearbox**, **Steering** objects, and many more components.

To build the **Car** class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

Encapsulation. Encapsulation conceals the functional details of a class from objects that send messages to it. For example, the **Car** class has a **accelerate()** method. The code for the **accelerate()** method defines exactly how a acceleration happens (with checking acceleration limits of car). Encapsulation is achieved by specifying which classes may use the members of an object.

The **reason for encapsulation** is to **prevent** clients of an interface **from depending** on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients.

Simple C# Program

C# demands that all program logic be contained within a type definition. Unlike many other languages, in C# it is not possible to create global functions or global points of data. Rather, **all data members and methods must be contained within a type definition.**

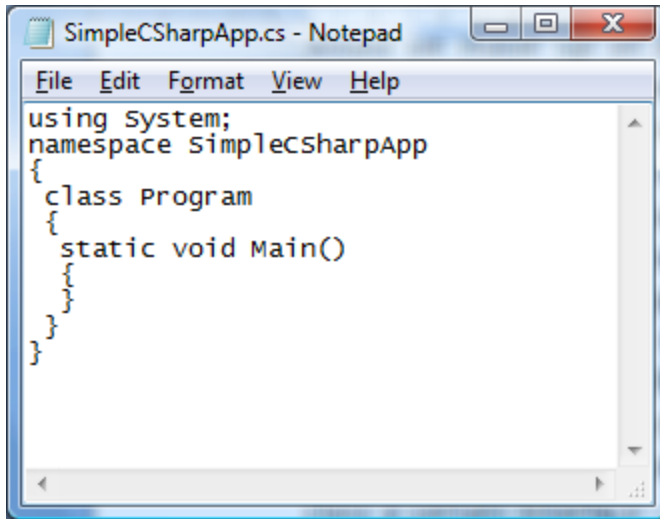
To get the ball rolling, create a new Console Application project named **SimpleCSharpApp**. As you might agree, the initial code statements are rather uneventful:

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
        }
    }
}
```

C# is a case-sensitive programming language. Therefore, “Main” is not the same as “main,” and “Readline” is not the same as “ReadLine.” Given this, be aware that **all C# keywords are lowercase** (public, lock, class, global, and so on), while **namespaces, types, and member names begin (by convention) with an initial capital letter and have capitalized the first letter of any embedded words.**

As a rule of thumb, whenever you receive a compiler error regarding
“undefined symbols” be sure to **check your spelling!**

Compilation and running



```
SimpleCSharpApp.cs - Notepad
File Edit Format View Help
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
        }
    }
}
```

1

You can create source code with Microsoft Notepad

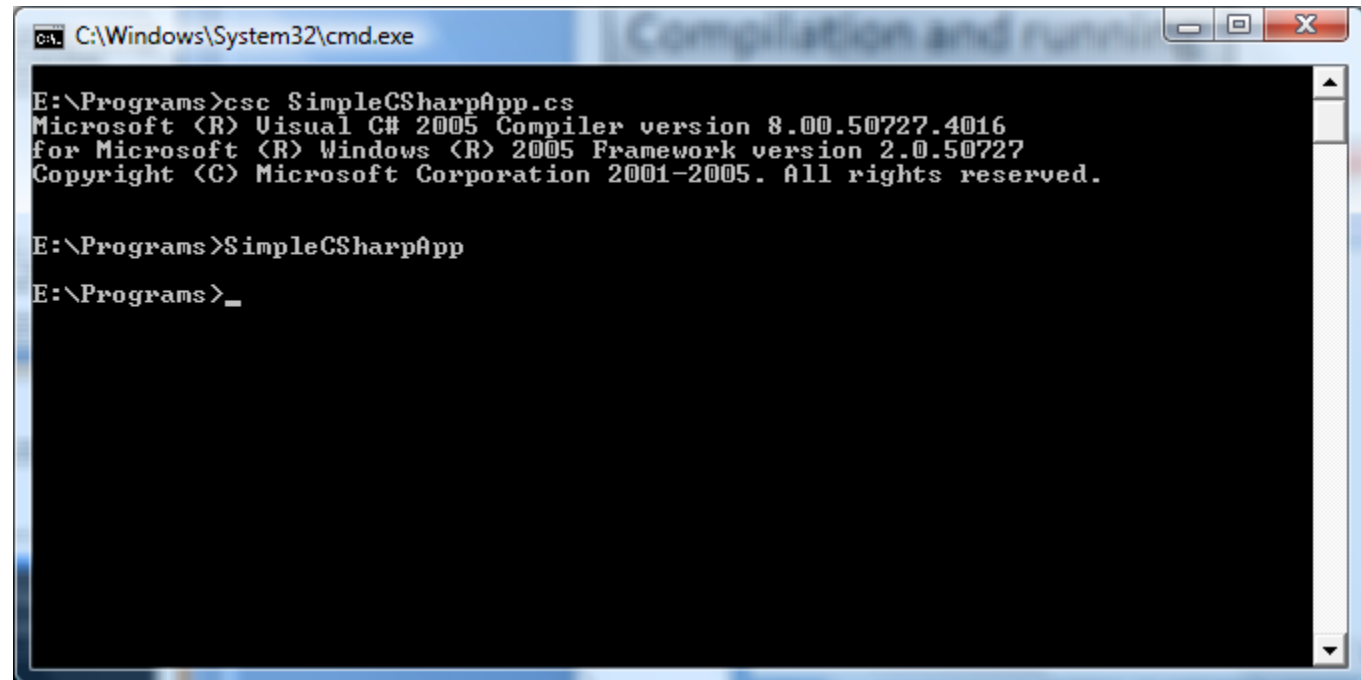
2

Compile your program with **csc.exe**

3

Run your program.

Result of running is nothing !!!



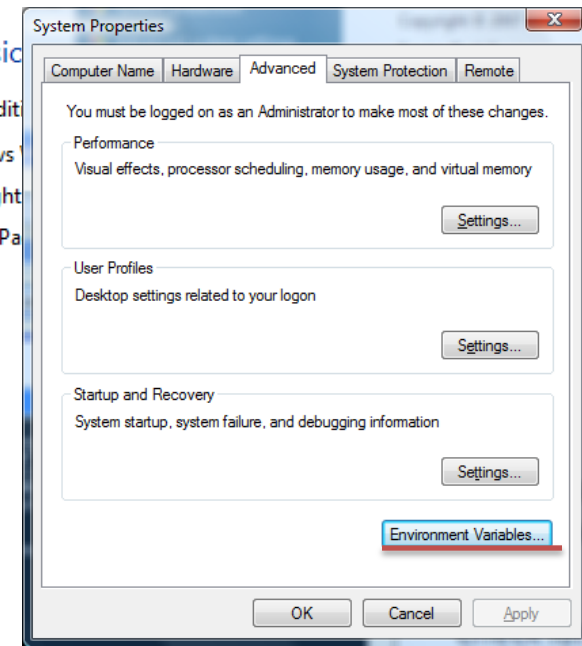
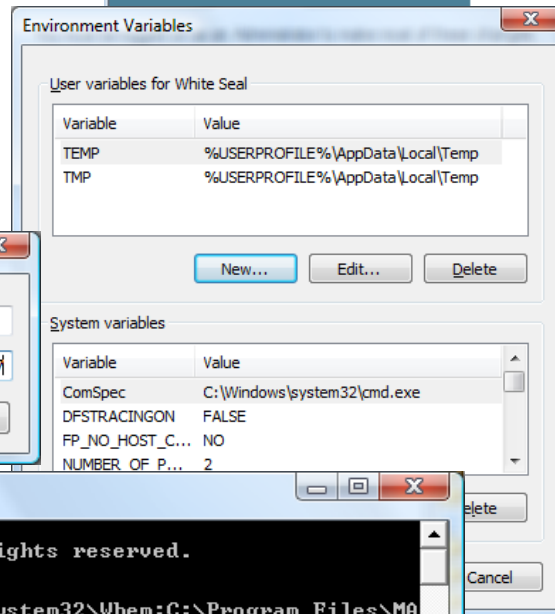
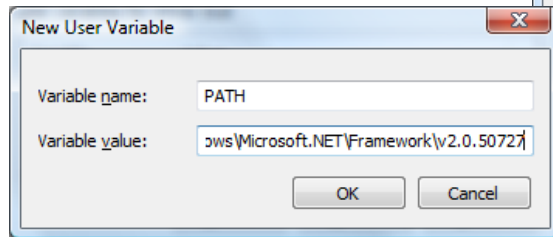
```
C:\Windows\System32\cmd.exe

E:\Programs>csc SimpleCSharpApp.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.4016
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

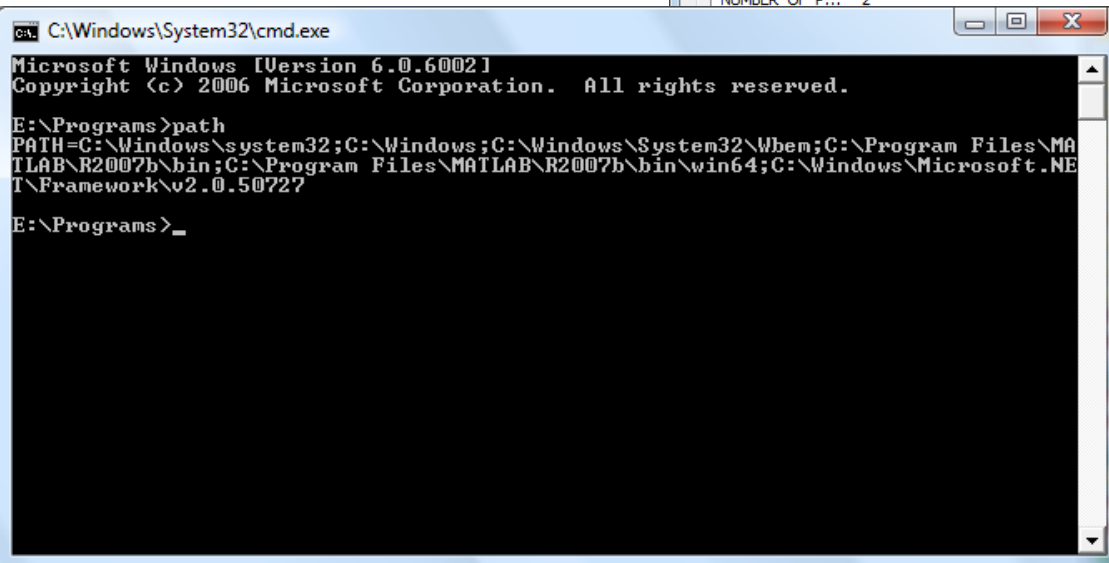
E:\Programs>SimpleCSharpApp
E:\Programs>_
```

Path specification

For comfort work with C# compiler you should specify .NET Framework path



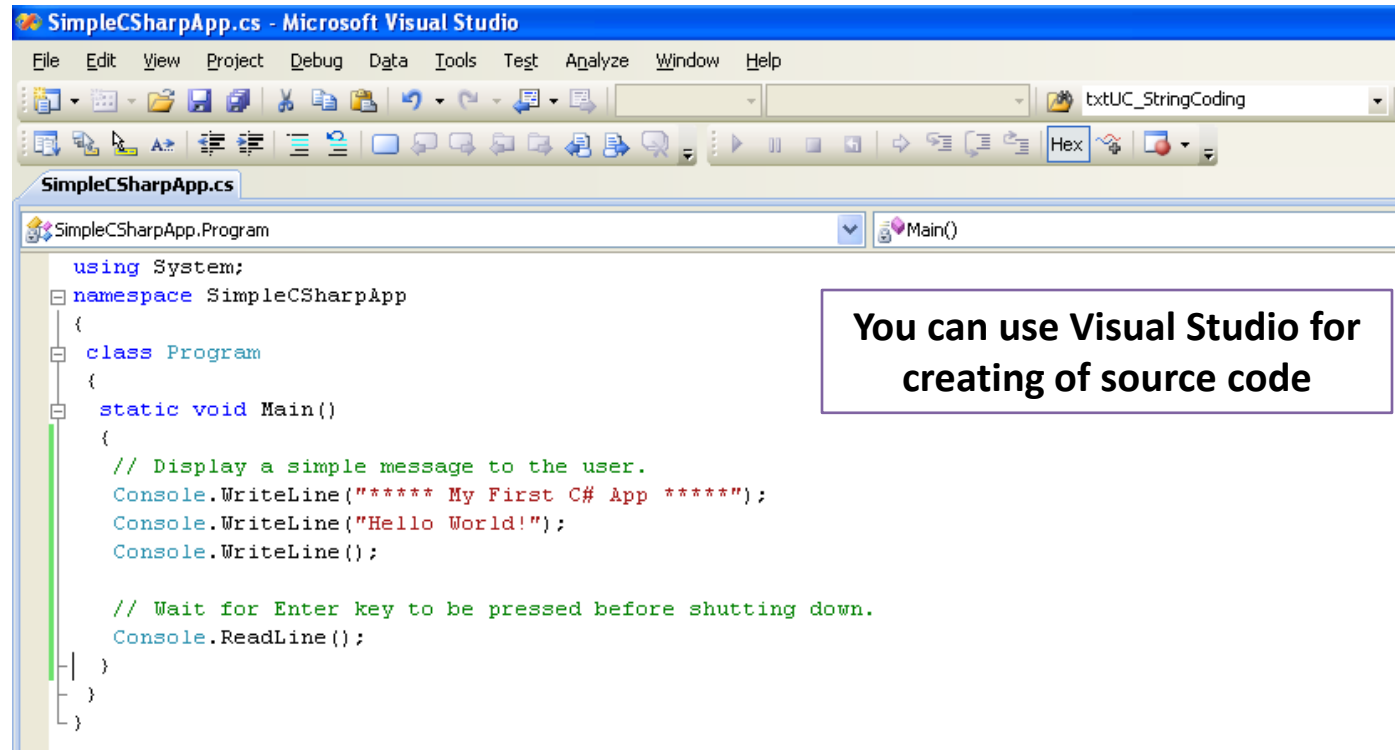
But you should have
Administrator privileges !!!
Check for changes by typing
path
in command window



Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

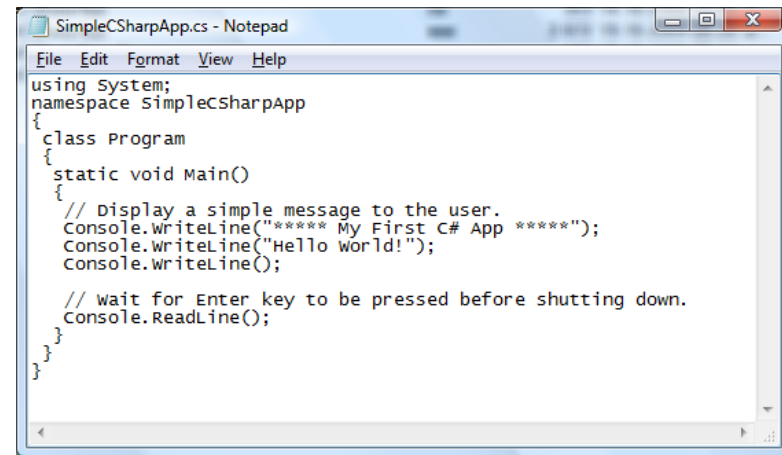
            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```

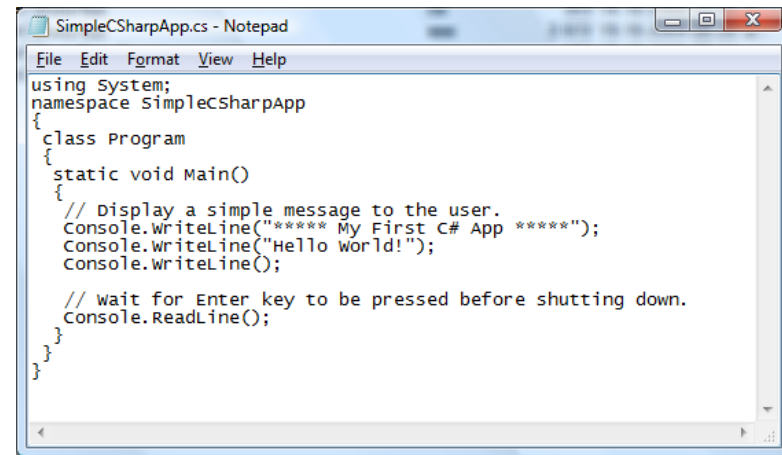


using System; This is an instruction to the C# compiler to tell it that we want to use things from the **System** namespace. A namespace is a place where particular names have meaning. In the case of C# the **System** namespace is where lots of useful things are described. One of these useful things provided with C# is the **Console** object which will let me write things which will appear on the screen in front of the user. If I want to just refer to this as **Console** I have to tell the compiler I'm using the **System** namespace. This means that if I refer to something by a particular name the compiler will look in.

Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



```
SimpleCSharpApp.cs - Notepad
File Edit Format View Help
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```

class Program A C# program is made up of one or more classes. A class is container which holds data and program code to do a particular job.

You need to invent an identifier for every class that you create. I've called ours **Program** since this reflects what it does. There is a convention that the name of the file which contains a particular class should match the class itself, in other words the program above should be held in a file called **Program.cs**.

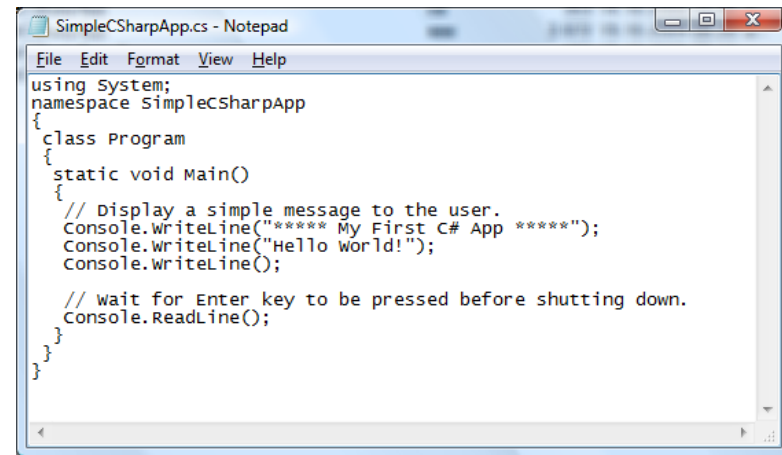
static This keyword makes sure that the method which follows is always present, i.e. the word static in this context means "it part of the enclosing class and is always here".

With using of static modifier, we declare, that program start point is Main method.

Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



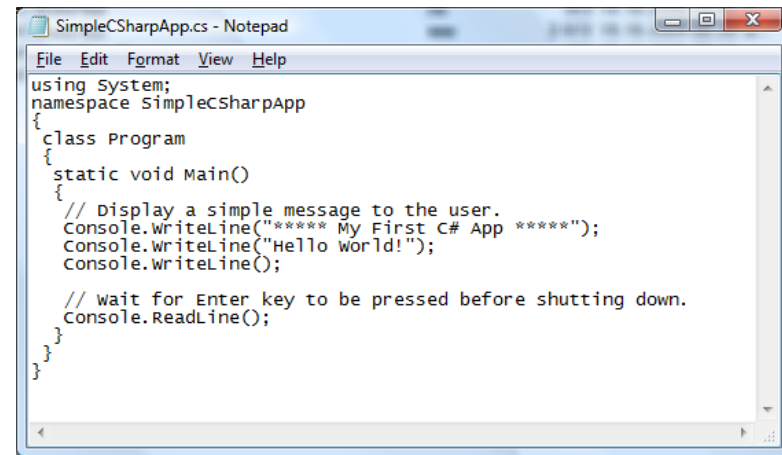
void A void is nothing. In programming terms the void keyword means that the method we are about to describe does not return anything of interest to us. The method will just do a job and then finish. In some cases we write methods which return a result.

Main You choose the names of your methods to reflect what they are going to do for you. Except for **Main**. This method (and there must be one, and only one such method) is where your program starts running. When your program is loaded and run the first method given control is the one called **Main**. If you miss out the **Main** method the system quite literally does not know where to start.

Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



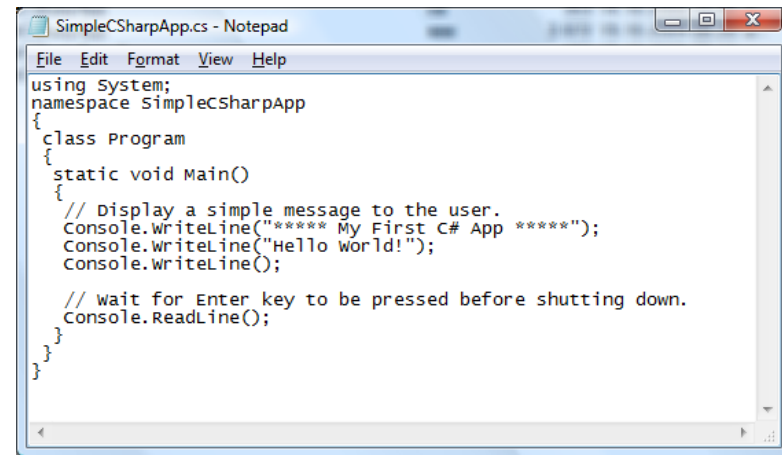
() This is a pair of brackets enclosing nothing. This may sound stupid, but actually tells the compiler that the method **Main** has no parameters. A parameter to a method gives the method something to work on. When you define a method you can tell C# that it works on one or more things, for example $\sin(x)$ could work on a floating point value of angle x .

{ and } This is a brace. As the name implies, braces come in packs of two, i.e. for every open brace there must be a matching close. Braces allow programmers to lump pieces of program together. Such a lump of program is often called a block. A block can contain the declaration of variable used within it, followed by a sequence of program statements which are executed in order. When the compiler sees the matching close brace at the end it knows that it has reached the end of the method and can look for another (if any). The effects of an unpaired brace are invariably fatal....

Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



// Comments Any good written program should have some comments. Any comment should be informative and describe the actions made/performed in lines of code.

; The semicolon marks the end of the list of variable names, and also the end of that declaration statement. All statements in C# programs are separated by the ; character, this helps to keep the compiler on the right track.

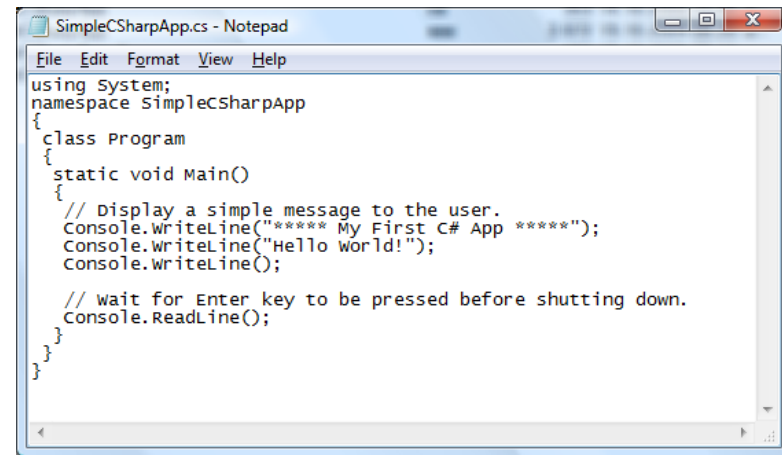
The ; character is actually very important. It tells the compiler where a given statement ends. If the compiler does not find one of these where it expects to see one it will produce an error.

Console It means that this method is part of an object called **Console** which looks after the user input and output.

Syntax explanation

```
using System;
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main()
        {
            // Display a simple message to the user.
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();

            // Wait for Enter key to be pressed before shutting down.
            Console.ReadLine();
        }
    }
}
```



The screenshot shows a Notepad window with the title 'SimpleCSharpApp.cs - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The code content is identical to the one in the previous block, showing a C# program that prints a message and waits for user input before exiting.

ReadLine This indicates that the ReadLine method is to be invoked. This asks the running program to dash off, do whatever statements there are in this method, and then come back.

The C# system contains a number of built in methods to do things for our programs. ReadLine is one of these. When this program runs the ReadLine method is invoked (or called). It will wait for the user to enter a line of text and press the Enter key. Whatever is typed in is then returned as a string by the ReadLine method.

() A method call is followed by the parameters to the method. A parameter is something that is passed into a method for it to work on. Think of it as raw materials for a process of some kind. In the case of ReadLine it has no raw materials, it is going to fetch the information from the user console. However, we still have to tell the method call something, even if it means there are no parameters being supplied.

Variations on the Main() Method

By default, Visual Studio will generate a Main() method that has a void return value and an array of string types as the single input parameter. This is not the only possible form of Main(), however. It is permissible to construct your application's entry point using any of the following signatures (assuming it is contained within a C# class or structure definition):

```
// int return type, array of strings as the argument.  
static int Main(string[] args)  
{  
}
```

```
// No return type, no arguments.  
static void Main()  
{  
}
```

```
// int return type, no arguments.  
static int Main()  
{  
}
```

Obviously, your choice of how to construct Main() will be based on two questions.

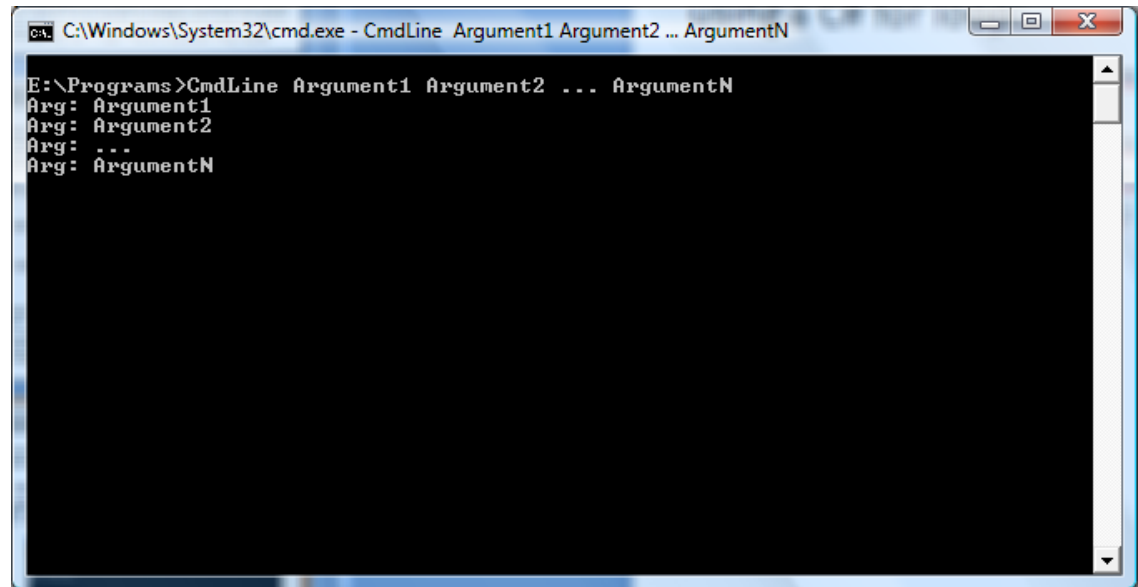
First, do you want to return a value to the system when Main() has completed and your program terminates? If so, you need to return an int data type rather than void.

Second, do you need to process any user-supplied command-line parameters? If so, they will be stored in the array of strings.

Processing Command-Line Arguments

let's examine the incoming array of string data. Assume that you now wish to update your application to process any possible command-line parameters. One way to do so is using a C# for loop:

```
static int Main(string[] args)
{
    // Process any incoming args.
    for(int i=0; i<args.Length; i++)
        Console.WriteLine("Arg: {0}", args[i]);
    Console.ReadLine();
    return -1;
}
```



Here, you are checking to see whether the array of strings contains some number of items using the Length property of **System.Array**.

Power of .NET system

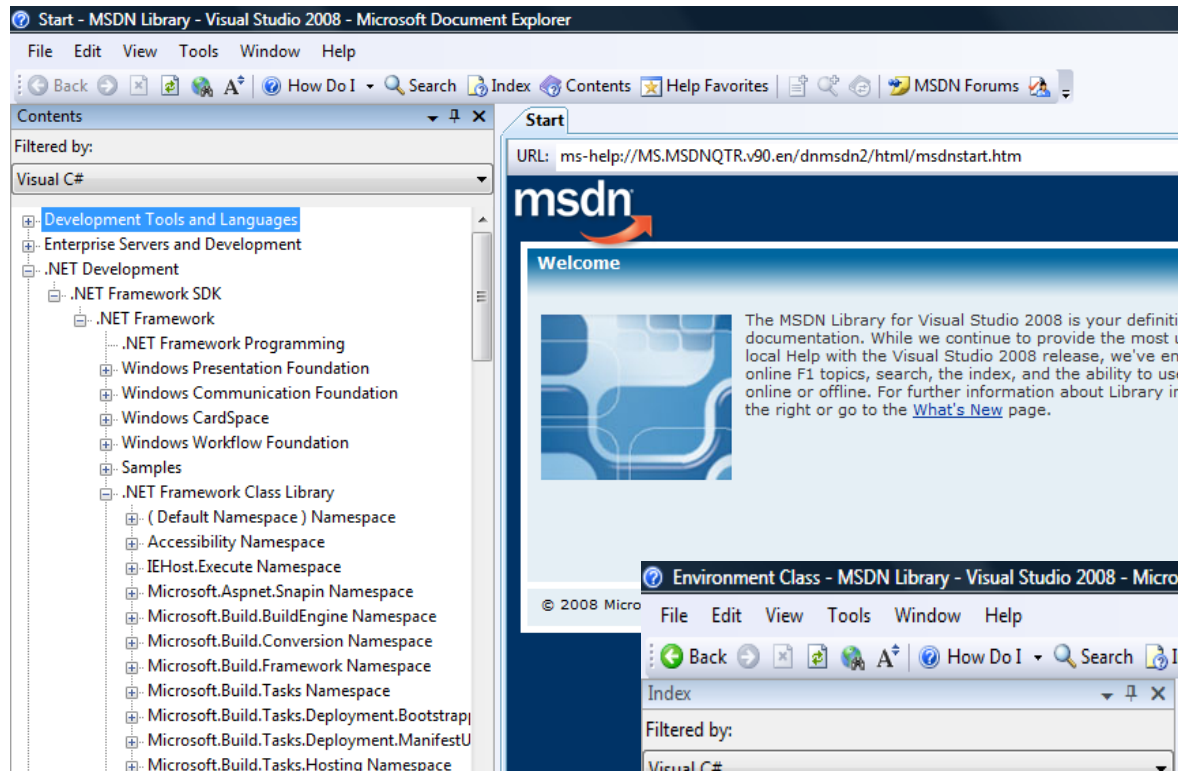
The `Environment` type exposes a number of extremely helpful methods beyond **`GetCommandLineArgs()`**. Specifically, this class allows you to obtain a number of details regarding the operating system currently hosting your .NET application using various static members. To illustrate the usefulness of `System.Environment`, update your `Main()` method to call a helper method named `ShowEnvironmentDetails()`:

```
static int Main(string[] args)
{
    ...
    // Helper method within the Program class.
    ShowEnvironmentDetails();
    Console.ReadLine();
    return -1;
}
```

Implement this method within your `Program` class to call various members of the `Environment` type. For example:

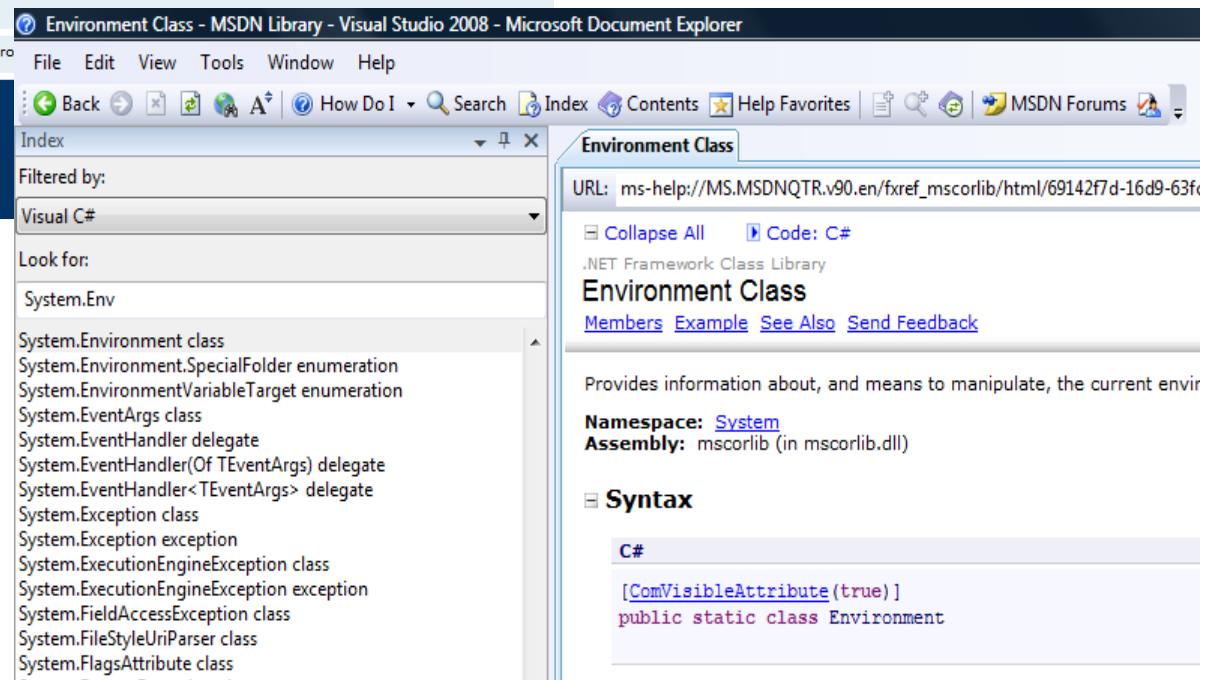
```
static void ShowEnvironmentDetails()
{
    // Print out the drives on this machine,
    // and other interesting details.
    foreach (string drive in Environment.GetLogicalDrives())
        Console.WriteLine("Drive: {0}", drive);
    Console.WriteLine("OS: {0}", Environment.OSVersion);
    Console.WriteLine("Number of processors: {0}", Environment.ProcessorCount);
    Console.WriteLine(".NET Version: {0}", Environment.Version);
}
```

Using of Microsoft Development Network



DVD-version

www.msdn.microsoft.com



Gathering information about Environment class possibilities

Property	Description
CommandLine	Gets the command line for this process.
CurrentDirectory	Gets or sets the fully qualified path of the current working directory.
ExitCode	Gets or sets the exit code of the process.
MachineName	Gets the NetBIOS name of this local computer.
NewLine	Gets the newline string defined for this environment.
OSVersion	Gets an OperatingSystem object that contains the current platform identifier and version number.
ProcessorCount	Gets the number of processors on the current machine.
StackTrace	Gets current stack trace information.
SystemDirectory	Gets the fully qualified path of the system directory.
TickCount	Gets the number of milliseconds elapsed since the system started.
UserDomainName	Gets the network domain name associated with the current user.
UserInteractive	Gets a value indicating whether the current process is running in user interactive mode.
UserName	Gets the user name of the person who is currently logged on to the Windows operating system.
Version	Gets a Version object that describes the major, minor, build, and revision numbers of the common language runtime.

Thanks for attention