

C# PROGRAMMING FOUNDATIONS:
Misc. Method/Object Issues



© August 2011 by
ERIC MATTHEWS

Copyright

C# Foundations by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unported License. This license allows you to

- Copy, distribute and transmit the work
- Adapt the work
- Make commercial use of the work

Under the following conditions:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to one or more of the following web pages.

www.DeveloperGeekResources.com

www.EducationAnytime.com

Contents

| | |
|---|-----------|
| Copyright | 2 |
| <i>About this document</i> | <i>4</i> |
| <i>C# Access specifiers</i> | <i>5</i> |
| Types | 5 |
| Protected | 5 |
| Public | 5 |
| Private | 5 |
| Internal | 5 |
| Using or Abusing or Not Using..... | 5 |
| In Action..... | 6 |
| <i>Initialization Issues</i> | <i>8</i> |
| How are variables initialized? | 8 |
| When does a constructor get called? | 8 |
| <i>Passing data in and out of Methods</i> | <i>11</i> |
| Using return | 11 |
| Passing arrays as an argument..... | 13 |
| Put into practice – Free Form | 16 |

ABOUT THIS DOCUMENT

This is a collection of topics that I just did not know where to fit into the other modules at the time I wrote them. Teaching programming classes a lot of times involves the chicken and the egg principle. You know that you really should cover a topic at a certain place, but you also know that it could lead you down a long and winding trail, or serve to confuse and bewilder your audience.

I have been teaching structured programming classes for over ten years and have found this to be the case. Trying to teach a class in Object Oriented Programming I have found only makes this dilemma larger.

These are certainly topics that need to be covered in a foundations course. I also wanted to allow some flexibility on the part of the teacher and felt that inclusion in a linear fashion would disrupt this goal.

I mention these things in case you try to use these documents outside of my class. I apologize that this the materials are not necessarily in a format that is all that linear. I am not sure that I will ever be able to get them into a linear format, as OOP is not exactly a linear subject.

As a final thought that might help you to use this document, I found that these materials are best covered after the module, "Second Steps in Object Oriented Programming".

C# ACCESS SPECIFIERS

First of all what is a C# access specifier? An access specifier determines what the consumer of your class can and cannot access. Sometimes you will here this access referred to as “member access”. Make note that a member refers to a class, a method, or a variable

TYPES

Protected

The type or member can be accessed only by code in the same class, or in a class that is derived from that class.

Public

The type or member can be accessed by any other code in the same assembly or another assembly that references it.

Private

The type or member can be accessed only by code in the same class.

Internal

The type or member can be accessed by any code in the same assembly, but not from another assembly.

USING OR ABUSING OR NOT USING

As a developer I am not a big fan of implementing too many controls in the development environment. My attitude greatly changes when we start talking about the production environment. But when it comes to this subject I have a few items to share.

When it comes to security how you mark your specifiers can have impact on security. There are a lot of factors involved in this particular subject and this is just not the time or place to dive into them.

Next, if you are going to be developing classes that other developers are going to use I have strong feelings that you should mark your classes appropriately. For example, if you have methods within a class that are not intended for other to directly access mark them as private! This is the whole point of encapsulation. Also, when you go to generation documentation these classes inherently will not get listed making it easier for the consumer to understand what you expect them to use. If they are directly

trolling the code the private declaration will be a visual queue as well that that class is not intended for their use.

When I was working in Java another development team had a number of api's we used within Blaze Advisor (a third party decision support engine). This team made everything public. Within Blaze advisor the programmer was exposed to hundreds of Java method of which they only needed to use a handful. I cannot tell you what a nightmare it made mentoring and technical turnovers.

IN ACTION

Here is a simple example of the discussion we just had. It can be found under the Access project.

Source Access1.cs

```
using System;

namespace Access
{
    public class Access1
    {
        public void My_interface()
        {
            Hello();
            Goodbye();
        }

        public void Hello()
        {
            Console.WriteLine("Hello");
        }

        private void Goodbye()
        {
            Console.WriteLine("Goodbye");
        }

    } //end class
} //end namespace
```

In this program we have three methods. One is declared as private. One of the methods is an “accessor method”, meaning its role is to call other methods in the class.

Take a look at the following program.

Source use_Access1.cs

```
using System;

namespace Access
{
```

```

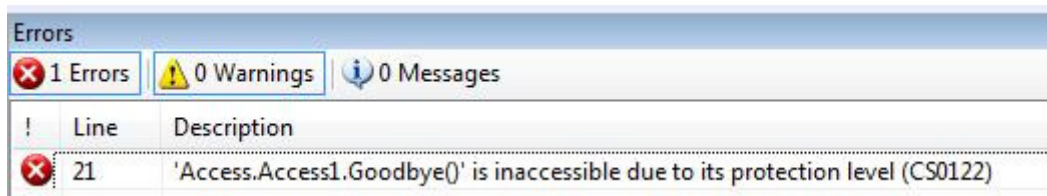
/// <summary>
/// Description of Class1.
/// </summary>
public class use_access1
{
    public static void Main(string[] args)
    {
        Access1 myaccess = new Access1();
        myaccess.My_interface();
        myaccess.Hello();
        // myaccess.Goodbye();
        Console.WriteLine("Press any key to continue . . . ");
        Console.ReadKey(true);
    }
}
}

```

We are able to access the two public methods by creating an instance of Access1. If you uncomment out the line...

```
// myaccess.Goodbye();
```

...you will receive the following error.



It can't access the method Goodbye() because it is private. Private gives you the ability to hide code from other consumers. This is what encapsulation is all about. These specifiers allow you the ability to hide code from the consumer, but to create interfaces that allow the consumer to use your code. Since C# is all about objects, the consumer is using your classes which is compiled into bytecode. Since our reuse effort is not with respect to the source code we really have insulated our code from others (well of course there are always disassemblers if someone is really intent on hacking our stuff). It is important to reiterate that reuse efforts take a great deal of thought and planning. Further, you need to be able to guarantee your methods when you encapsulate your code. Meaning... What is a programmer to do if they are using your stuff and suddenly it breaks their stuff because you have made changes?

A couple of side notes worth mentioning here. This project has two files associated with it. Each files represents a class under the namespace named Access. As mentioned earlier a namespace is a logical wrapper that can hold multiple classes. The Access1 class represents our api (lame as it is) that other classes can use. Use_Access1 is the class that is using the Access1.class. It contains our main method. It should be noted that in a C# project that has multiple classes you can only have one main method or you will receive the compile error ".exe has more than one entry point."

INITIALIZATION ISSUES

HOW ARE VARIABLES INITIALIZED?

The answer to this question depends on where they are declared. Consider the following program.

source Vinit1.cs

```
using System;

class Vinit1
{
    public static int a;
    public static void Main(string[] args)
    {
        Console.WriteLine(a);
        a=5;
        Console.WriteLine(a);
        int b;
        // b=0;
        Console.WriteLine(b);

        Console.Write("Press any key to continue . . . ");
        Console.ReadKey(true);
    }
}
```

This program will not compile as it is written. We receive a compiler error.



The real question is not why did we receive a compiler error for not initializing “b”, rather why did we not receive an error for not initializing “a” as well? “a” is a primitive data type and a direct part of the class itself. “a” gets initialized automatically by the default constructor. What this means is that if you say “a=0;” in your code that “a” gets initialized twice; once by your statement, and once by the default constructor. The constructor always exists in a class, even if you do not directly refer to it in your code.

WHEN DOES A CONSTRUCTOR GET CALLED?

A constructor is called only after all the variables and references exposed to the class are initialized. By variables I am referring to both primitives and handles (references to classes). The next example should clarify what I just said.

source Caller.cs

```
using System;

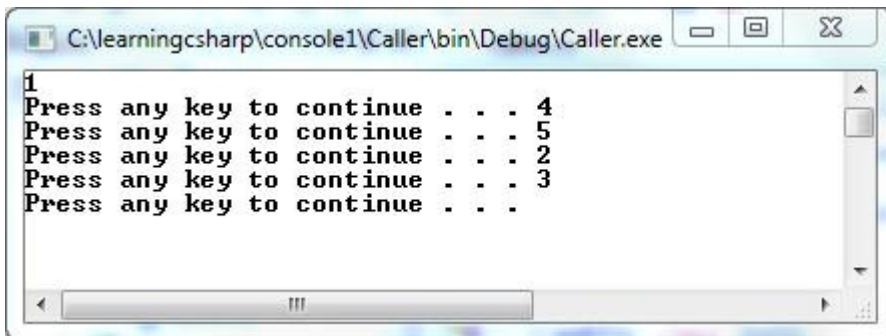
public class Caller
{
    public static void Main(String[] args)
    {
        CallInit1 I = new CallInit1();
    }
}

public class Init1
{
    public Init1(int i)
    {
        Console.WriteLine(i);
        Console.Write("Press any key to continue . . . ");
        Console.ReadKey(true);
    }
}

public class CallInit1
{
    Init1 I1 = new Init1(1);

    public CallInit1() // constructor
    {
        Init1 I2 = new Init1(2);
        Init1 I3 = new Init1(3);
    }

    Init1 I4 = new Init1(4);
    Init1 I5 = new Init1(5);
}
```



This makes good sense. We do not want to initialize our class until we have created all the objects and storage that it requires.

PASSING DATA IN AND OUT OF METHODS

The following program will give a good perspective on passing values into a method, and returning a result. Review the code then read the walkthrough that follows.

Additionally, be aware that arguments are passed to methods by value. This means that a copy is made for the argument we are passing. And while I am thinking about it I realize that we always use the term passing to describe this operation and that this is really a poor choice of words. If I pass you a football, this means that you now have the football, and I do not. In computer science when I pass a parameter by value, I am essentially making a clone of the football and sending you the clone. If I pass you the parameter by reference I am not really giving you the football, I am sending you a pointer to where I have the football.

Anyhow, it is important that you understand that we pass by value. The ensuing code and discussion will demonstrate passing various primitives and objects into and out of methods.

USING RETURN

Source Return1.cs

```
using System;

public class Return1
{
    public float mult(float i,float y)
    {
        float x=0;
        x=(i*y);
        return x;
    }

    public float div(float i,float y)
    {
        float x=0;
        if (y != 0)
        {
            x=(i/y);
        } else
        {
            Console.WriteLine("Divide by 0");
        }
        return x;
    }

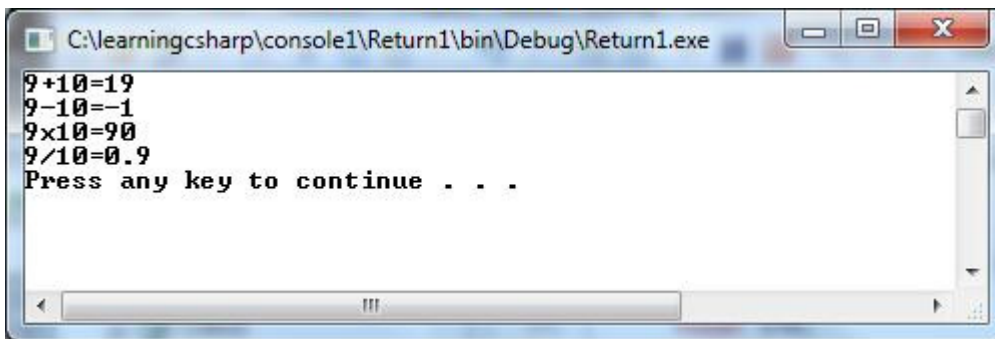
    public float add(float i,float y)
    {
        float x=0;
        x=(i+y);
        return x;
    }
}
```

```
}

public float subtr(float i,float y)
{
    float x=0;
    x=(i-y);
    return x;
}

public static void Main(string[] args)
{
    float r=0;
    float g=9;
    float h=10;
    Return1 c = new Return1();
    r=c.add(g,h);
    Console.WriteLine(g + "+" + h + "=" + r);
    r=c.subtr(g,h);
    Console.WriteLine(g + "-" + h + "=" + r);
    r=c.mult(g,h);
    Console.WriteLine(g + "x" + h + "=" + r);
    r=c.div(g,h);
    Console.WriteLine(g + "/" + h + "=" + r);

    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
}
}
```



Lets first look at the methods. Since they are all pretty much the same we will examine the code for `div()`.

```
public float div(float i,float y)
{
    float x=0;
    if (y != 0)
    {
        x=(i/y);
    } else
    {
        Console.WriteLine("Divide by 0");
    }
}
```

```
return x;
}
```

The `div()` method accepts two arguments. Both are primitive data of type `float`. It is important to remember that we are actually passing by value, not by reference. Variables “`i`” and “`y`” are local to the method. While changing the values of either of these will affect the outcome of what “`x`” returns they have not impact whatsoever on the variables that are used by the caller. Our method `div()` will return float “`x`” to the caller.

Back to our main program...

```
float r=0;
float g=9;
float h=10;
Return1 c = new Return1();
```

... we declare and set the variables we need in order to do the method call. Then, we create a `Return1` object. Once we have created this object we have access to the methods (it allows us exposure to, which in this case is all of them).

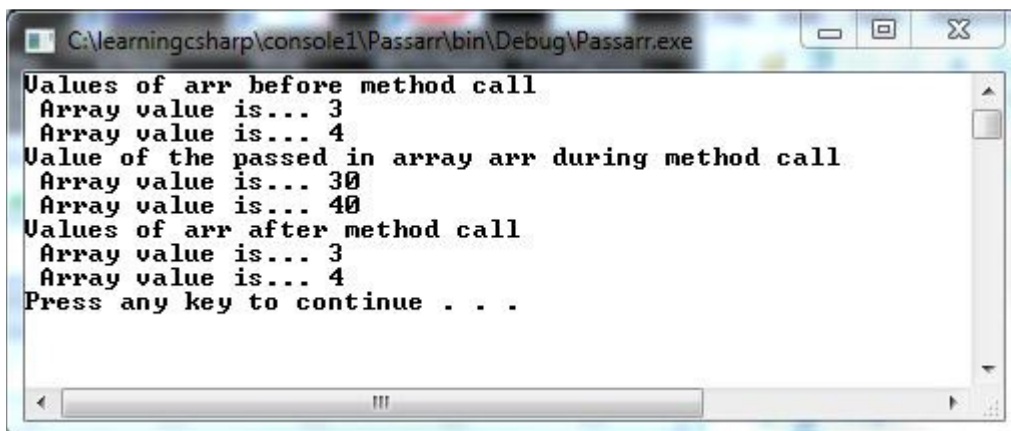
```
r=c.div(g,h);
Console.WriteLine(g + "/" + h + "=" + r);
```

We call the `div()` method as part of an expression. We are looking through the assignment operator for the method to return a value to our variable “`r`”. We call the method on the other side of the assignment operator. Notice we separate what we are passing with the call using the comma (,) operator. Position is important. The method will return “`x`” to “`r`”. I purposely used different variable names so that you could see that these variables too are indeed different. Because of how namespaces are used in C# we could have used the same names for all of our variables.

It is also possible to pass arrays and for that matter, any objects of any type into a method. It is also possible to return such items as part of a method call.

PASSING ARRAYS AS AN ARGUMENT

When you pass an array by argument how does the array get passed? By reference? By value? You have to be careful asking such a question, or at minimum understand that the value of an array is inherently a reference. Yes this is confusing, but keep reading and I will try to explain.



```
C:\learningcsharp\console1\Passarr\bin\Debug\Passarr.exe
Values of arr before method call
Array value is... 3
Array value is... 4
Value of the passed in array arr during method call
Array value is... 30
Array value is... 40
Values of arr after method call
Array value is... 3
Array value is... 4
Press any key to continue . . .
```

Source Passarr.java

```
using System;

public class Passarr
{
    public static void Main(String[] args)
    {
        int[] arr = {3,4};
        Console.WriteLine("Values of arr before method call");
        disp(arr);
        acceptarr(arr);
        Console.WriteLine("Values of arr after method call");
        disp(arr);

        Console.Write("Press any key to continue . . . ");
        Console.ReadKey(true);
    }

    public static void acceptarr(int[] arr)
    {
        int[] d = {30,40};
        arr=d;
        Console.WriteLine("Value of the passed in array arr during method call");
        disp(arr);
    }

    public static void disp(int[] a)
    {
        Console.WriteLine(" Array value is... " + a[0]);
        Console.WriteLine(" Array value is... " + a[1]);
    }
}
```

Here is what is happening. First we create an array named “arr” and place two values in it (3,4). Next we pass this array into a method named “acceptarr”. We assign the array we passed to another array named “d” in our method. We then check the values of the input argument arr and they are (30,40). After the method call we display the contents of the array we passed and see that the values are still (3,4).

So, we can conclude that what has occurred is that we are working on a copy of the array that was passed into acceptarr(). If we want to actually change the array we are passing here is what we need to do.

Source Passarr2.java

```
using System;

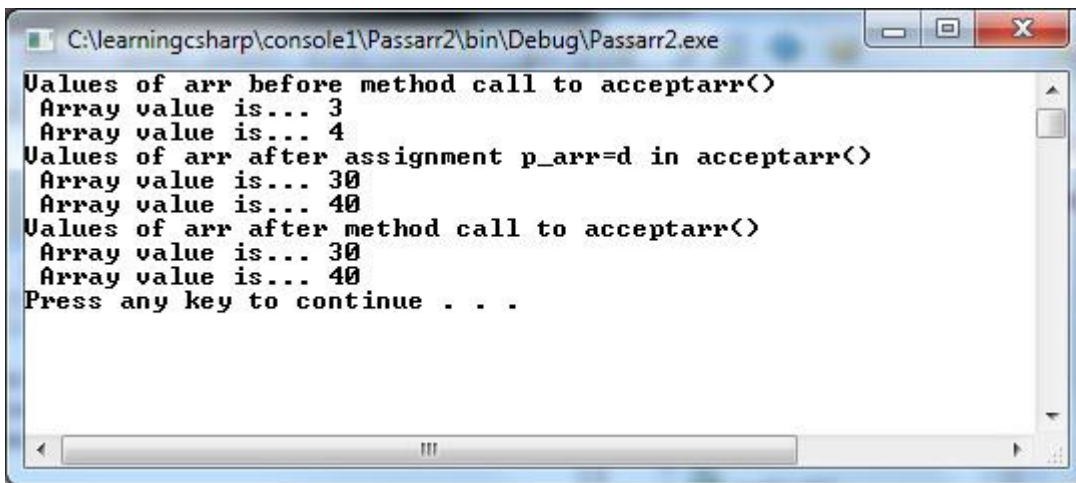
public class Passarr2
{
```

```
public static void Main(string[] args)
{
    int[] arr = {3,4};
    Console.WriteLine("Values of arr before method call to acceptarr()");
    disp(arr);
    arr = acceptarr(arr);
    Console.WriteLine("Values of arr after method call to acceptarr()");
    disp(arr);

    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
}

public static int[] acceptarr(int[] arr)
{
    int[] d = {30,40};
    p_arr=d;
    Console.WriteLine("Values of arr after assignment p_arr=d in acceptarr()");
    disp(p_arr);
    return p_arr;
}

public static void disp(int[] a)
{
    Console.WriteLine(" Array value is... " + a[0]);
    Console.WriteLine(" Array value is... " + a[1]);
}
}
```



```
C:\learningcsharp\console1\Passarr2\bin\Debug\Passarr2.exe
Values of arr before method call to acceptarr()
Array value is... 3
Array value is... 4
Values of arr after assignment p_arr=d in acceptarr()
Array value is... 30
Array value is... 40
Values of arr after method call to acceptarr()
Array value is... 30
Array value is... 40
Press any key to continue . . .
```

In this case we are changing the array that was passed to us only because we are returning it back. Note the caller function is expecting a return argument. In this case it is the object we passed in.

```
arr = acceptarr(arr);
```

PUT INTO PRACTICE – FREE FORM



Play around creating your own classes and playing with existing projects to get comfortable with the material covered in this module.