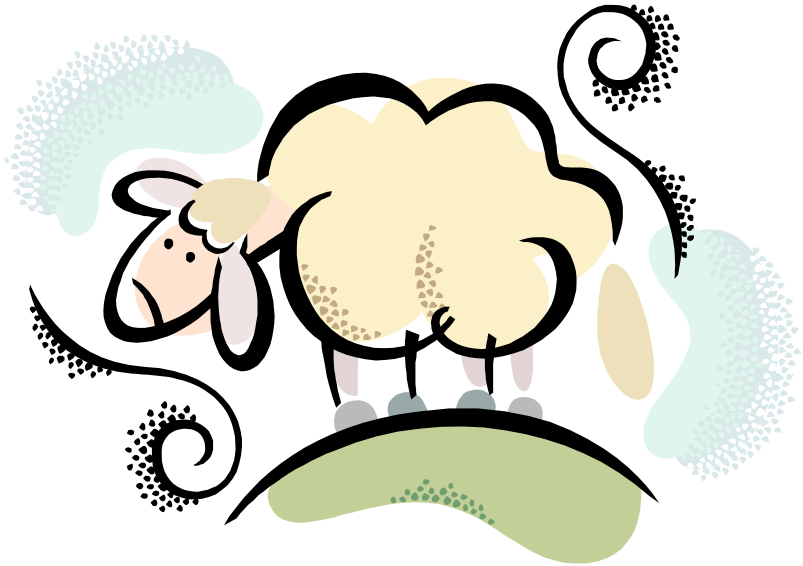


C# Programming II:

Beginning C#

Legal Stuff

This presentation as well as other presentations and documents found on the sheepsqueezers.com website may contain quoted material from outside sources such as books, articles and websites. It is our intention to diligently reference all outside sources. Occasionally, though, a reference may be missed. No copyright infringement whatsoever is intended, and all outside source materials are copyright of their respective author(s).

.NET Lecture Series

	C# Programming I: Concepts of OOP	*C# Programming II: Beginning C#*	*C# Programming III: Advanced C#*	
C# Programming IV-1: System Namespace	*C# Programming IV-2: System.Collections Namespace*	*C# Programming IV-3: System.Collections. Generic Namespace*	*C# Programming IV-4A: System.Data Namespace*	*C# Programming IV-4B: System.Data.Odbc Namespace*
C# Programming IV-4C: System.Data.OleDb Namespace	*C# Programming IV-4D: Oracle.DataAccess.Client Namespace*	*C# Programming IV-4E: System.Data.SqlClient Namespace*	*C# Programming IV-4F: System.Data.SqlTypes Namespace*	*C# Programming IV-5: System.Drawing/(2D) Namespace*
C# Programming IV-6: System.IO Namespace	*C# Programming IV-7: System.Numerics*	*C# Programming IV-8: System.Text and System.Text. RegularExpressions Namespaces*	*C# Programming V: Introduction to LINQ*	

C# Self-Inflicted Project #1

Address Cleaning

C# Self-Inflicted Project #2

Large Intersection Problem

Charting Our Course

- ❑ Pre-Requisites
- ❑ Our First Program…How Lovely!
- ❑ .NET Data Types
- ❑ Control Structures
- ❑ Casting Between Types
- ❑ Classes
- ❑ Constructors
- ❑ Smart Fields
- ❑ Derived Classes
- ❑ Static Classes
- ❑ Operator Overloading
- ❑ Structures Are Not Dead!
- ❑ Arrays
- ❑ Enumerated Types
- ❑ Handling Exceptions
- ❑ The `is` and `as` Operators
- ❑ C# Compiler Command Line Arguments
- ❑ What Next?

Pre-Requisites

Before you begin this lecture, please make sure that you have looked through the first lecture *C# Programming I*.

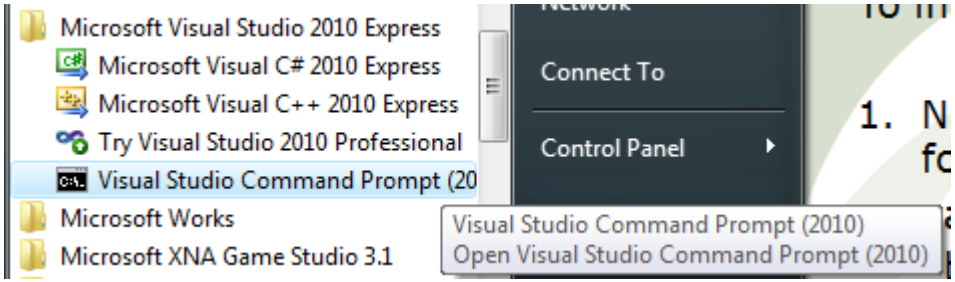
Also, you will need to download and install both Microsoft Visual C# 2010 Express as well as Microsoft Visual C++ 2010 Express. The reason you will need both is that the C++ edition installs a shortcut which opens a command window allowing you to compile your C# (as well as C++) programs at the command line. Microsoft Visual C# 2010 Express only allows you to program via Visual Studio. We will not program with Visual Studio until later in the lecture series since I feel it is more important to concentrate on learning the C# language first rather that the GUI interface…I laugh in the face of point-and-click…ha-ha!

To install these programs, perform the following steps:

1. Navigate your browser to the Microsoft/Express website. You will see entries for Visual C# 2010 Express as well as Visual C++ 2010 Express.
   a. Click on Visual C++ 2010 Express.
   b. Select a language. You will be prompted to save the file `vc_web.exe` to your hard drive.
   c. Double-click on this file to start the download and follow the instructions throughout.
2. Repeat instructions 1a, 1b and 1c for Visual C# 2010 Express.

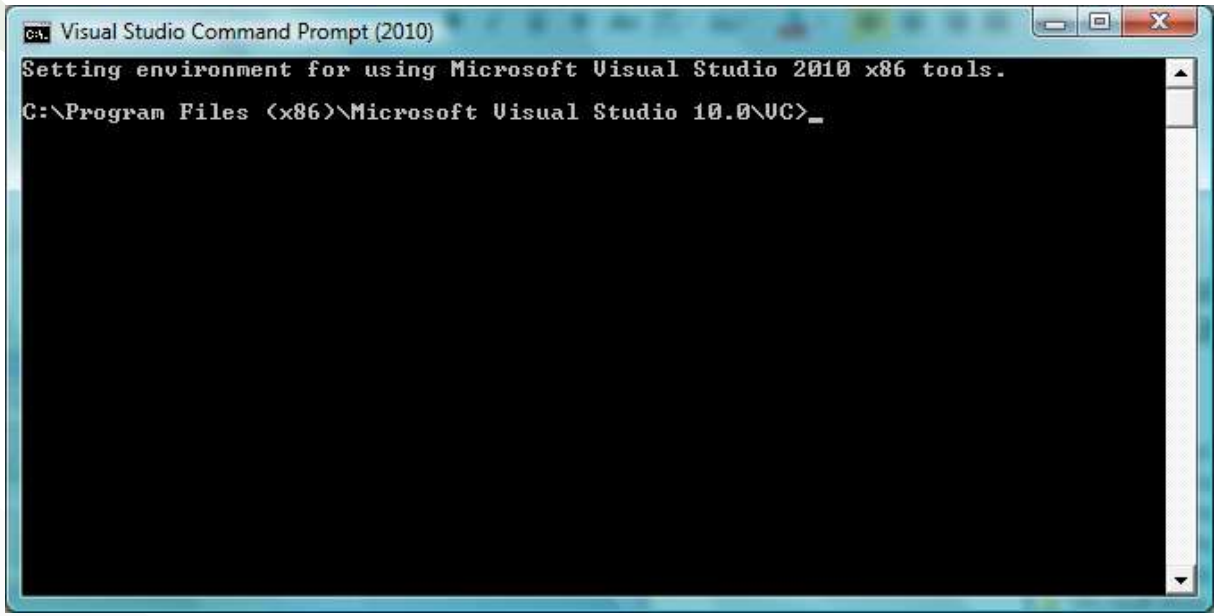
Pre-Requisites

Once you have completed installing both C# and C++ on your computer, you will see the following entries in the menu Start…All Programs…Microsoft Visual Studio 2010 Express:



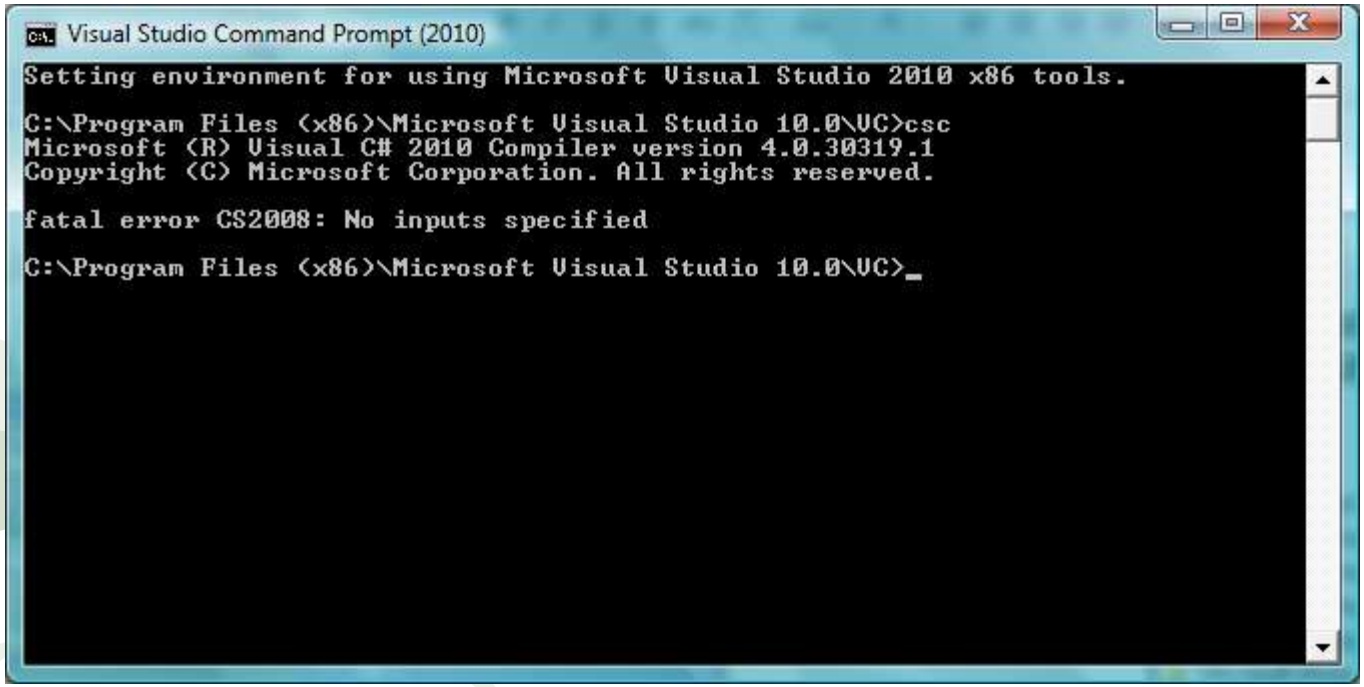
As you see, both Visual C# 2010 Express and Visual C++ 2010 Express have been installed, but you will also note that the Visual Studio Command Prompt shortcut has been installed as well. We will spend most of our time in this lecture in this window.

When you click on this shortcut, you will see the following very exciting window:



Pre-Requisites

If you type in `csc` at the command prompt and hit enter, you will see the following:



This indicates that the C# compiler, `csc`, is installed and ready to use…woo-hoo!

Note that if you open up an ordinary command window and type in `csc`, you will be told that the program `csc` cannot be found. The reason is that the Visual Studio Command Prompt shortcut, as described on the previous slide, sets up several environment variables as well as updates the path to ensure that the C# compiler can be found. The file `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\vcvarsall.bat` is executed when you click on the Visual Studio Command Prompt shortcut.

Our First Program…How Lovely!

Before we get into the details of C#, let's take a look at how to compile a small C# program. Open up your favorite editor such as Notepad, TextPad, UltraEdit, etc. and enter in the following C# program:

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  Console.WriteLine("Can someone help me clean my feet?\n");

 }

}
```

Next, save your program as `test1.cs` in a folder of your choosing. (My choosing is C:\TEMP\TEST, but that's because I live life on the edge…tee-hee…). At the Visual Studio Command Prompt, change to your folder, type in `csc test1.cs` and hit enter. If all went well, your program compiled with no errors.

Next, to execute your program, type in `test1.exe` (or just `test1`) at the command prompt and hit enter. You should see the words `Can someone help me clean my feet?` appear.

Hooray! We've just compiled and executed our very first, yet totally boring, C# program. Let's go through the code line by line.

Our First Program…How Lovely!


```
1  using System;

2  class OurFirstProgram {

     //Main program
3    public static void Main() {

4     Console.WriteLine("Can someone help me clean my feet?");

     }

   }
```

1. In order to access built-in methods such as `Console.WriteLine`, we need to tell C# which *namespaces* – a collection a methods (we discuss namespaces more later on in the lecture) – we are going to use. In this case, the System namespace is being used since it contains the method `Console.WriteLine`.
2. Since .NET is object-oriented, even our main program needs to be inside a class. In this case, I just give our class the name `OurFirstProgram`. In my opinion, most of the work will be done in other classes, but the whole ball gets rolling from within the class that contains the `Main` program.
3. Here is where we define our `Main` program. This is what Windows runs when we type in `test1.exe` at the command prompt. Note that all main programs must be `public` and `static`. We talk more about those later. If you do not need to return a return code, then `void` is the way to go; otherwise, use another type.
4. The method `Console.WriteLine()` writes out the text supplied in the parameter to the console window. Note that `WriteLine` automatically leaves a blank line, so there is no need for a carriage-return/linefeed character.

Our First Program…How Lovely!

Just like every other programming language that has been invented, C# programs can create and use variables. We talk more about the .NET data types later on, but here is how to create a string variable holding the text *Can someone help me clean my feet?*.

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  //Create a variable to hold our text
  String sTxt = "Can someone help me clean my feet?";

  Console.WriteLine(sTxt);

 }

}
```

1. As you can see, we use the `String` data type to create a string called `sTxt` which is then used in the `Console.WriteLine()` method. This brings us to a discussion about data types in .NET.

.NET Data Types

In most, if not all, of the programming languages you've learned, you've dealt with variables containing items such as numbers and strings. Numbers can be integers or floating point numbers. Strings are just a set of one or more characters. Numbers and strings are called *data types*. Since these data types are, in fact, hard-wired in the compiler itself, they are also called *primitives* or *primitive data types*. Data types such as integers, longs, floating point numbers, strings, booleans, and so on are examples of primitive data types.

Some languages allow for more complex types such as arrays, dictionaries, and so on. While you may have access to these data types, they are not part of the compiler and, thus, not primitive data types. These data types are built-up from primitive data types. For example, an array is a numbered collection of, say, integers which itself is a primitive data type.

The .NET Framework Class Library(FCL) contains thousands of type definitions beyond just the mundane integer and string. Because the FCL contains so many types, they are grouped together logically into *namespaces*. An example of a namespace is the `System` namespace we coded as the first line in our test program. This particular namespace holds the basic data types listed on the next slide.

Other namespaces are `System.Data`, which allows for communication with a database; `System.IO`, which allows for file and directory I/O; and `System.Text`, which allows you to work with text.

.NET Data Types

Primitive Type Name	FCL Type Name	C# Alias	Description
sbyte	System.SByte	SByte	Signed 8-bit byte (-2^7 to 2^7-1)
byte	System.Byte	Byte	Unsigned 8-bit byte (0 to 2^8-1)
short	System.Int16	Short	Signed 16-bit (-2^{15} to $2^{15}-1$)
ushort	System.UInt16	Ushort	Unsigned 16-bit (0 to $2^{16}-1$)
int	System.Int32	Int	Signed 32-bit (-2^{31} to $2^{31}-1$)
uint	System.UInt32	Uint	Unsigned 32-bit (0 to $2^{32}-1$)
long	System.Int64	Long	Signed 64-bit (-2^{63} to $2^{63}-1$)
ulong	System.UInt64	Ulong	Unsigned 64-bit (0 to $2^{64}-1$)
char	System.Char	Char	16-bit Unicode character
float	System.Single	Float	IEEE 32-bit float ($+/-10^{-45}$ to $+/-10^{38}$)
double	System.Double	Double	IEEE 64-bit float ($+/-10^{-324}$ to $+/-10^{308}$)
bool	System.Boolean	Bool	Boolean value of **true** or **false**.
decimal	System.Decimal	Decimal	128-bit positive/negative data type exact to 28 or 29 digits (except during a financial crisis…that's just me editorializing…sorry…)
object	System.Object	Object	Base class for all Common Type System (CTS) types
string	System.String	String	An array of characters.

So, which do you use in your code? Since you will always include `using System;` in your code, you can use the second column without the `System.`. This is recommended by one author (Jeffrey Richter). Thus, use `Int32` instead of `int` or `Int`.

.NET Data Types

Now, another important thing to know is that Common Language Runtime(CLR) splits up the FCL into two distinct types: *reference types* and *value types*.

The difference between the two is where the data is stored. Data can be stored on the *thread's stack* or on the *managed heap*. The managed heap is a block of memory which is under to control of the CLR garbage collector. If too much managed heap is being used, the garbage collector removes unneeded data. We talk more about garbage collection later on.

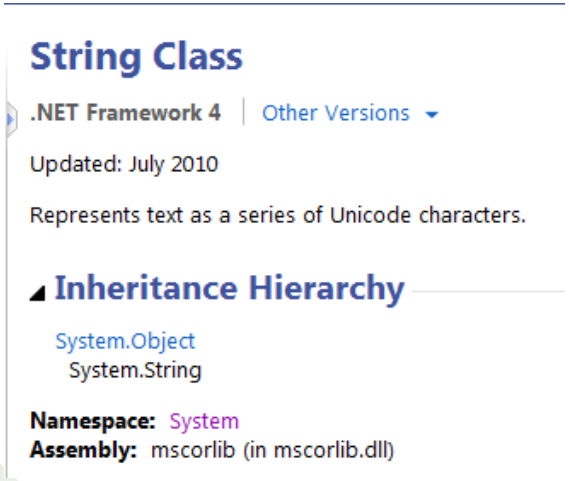
For value types, such as `Int32`, etc., values are allocated on the thread's stack. You can think of value types as the *normal* way of working with variables you are familiar with. A value type is never null and must contain a value, even if it's zero or blank. Value types are *not* under control of the garbage collector.

Reference types, such as `String`s, are allocated on the managed heap and an address location is returned instead of the actual value. (This is similar to the address operator (&) in C.) A reference type can be null, but if it's not null you are guaranteed that it points to an object of the type you've specified in your program.

Most of the types in the FCL are reference types, and most of the types we programmers use on a daily basis are value types. Value types are faster to create and access than reference types.

.NET Data Types

Note that in the .NET documentation, a reference type is referred to as a *class* whereas a value type is referred to as either a *structure* or *enumeration*. For example, `String`s are reference types as you can see from the documentation (note that it is referred to as `String` Class):



String Class

.NET Framework 4 | Other Versions

Updated: July 2010

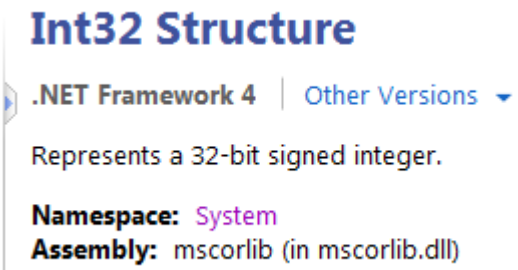
Represents text as a series of Unicode characters.

Inheritance Hierarchy

System.Object
  System.String

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

On the other hand, an `Int32` is a value type since it is referred to as an `Int32` Structure:



Int32 Structure

.NET Framework 4 | Other Versions

Represents a 32-bit signed integer.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

You can find these on the .NET System namespace web page at: http://msdn.microsoft.com/en-us/library/yxcx7skw.aspx.

.NET Data Types

Something you may have noticed in the table of data types is the `System.Object` class. The reason this is there is because ALL objects in .NET derive solely from the base class `System.Object`. Thus, .NET is referred to as a *singly rooted hierarchy*. Note that even if you code classes without explicitly specifying `System.Object`, it is put there automatically. So there!

So, what does this mean? In fact, surprising as it may seem, even numbers like 42 are just objects to .NET. For example,

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  Console.WriteLine(42.ToString());

 }

}
```

As you see above, we are executing the `ToString()` method of the number 42. (No variables were harmed during the programming of this example.)

Now, because of this singly rooted hierarchy, ALL objects in .NET have a minimum set of methods as shown on the next slide:

.NET Data Types

Method Name	Description
bool Equals	Compares two object references and if the two variables refer to the same object, true is returned. With two value types, true is returned if the two types are identical and have the same type.
int GetHashCode	Returns the hash code specified for an object.
Type GetType	Used with reflection methods to retrieve the type info for the object.
string ToString	Returns the name of the object
void Finalize	Called before the garbage collector cleans up.
Object MemberwiseClone	Represents a shallow copy of the object.

The first four methods are `public` methods and are available no matter how big your bollocks are…oh, I'm sorry…*public*, not *pubic*…my mistake…tee-hee…

The last two methods are `protected` methods and are available to derived classes. (I've dispensed with the very humorous pubic/public comment involving *protected* herewith). We talk more about `Finalize` and `MemberwiseClone` later on.

One additional note is that when if you initialize your variables with certain numeric values, you may need to use an appropriate *suffix* after the value. For `Int64`, the suffix is `L`; for `UInt32`, use `U`; for `UInt64`, use `UL`; for `Single`, use `F`; for `Double`, use `D`; for `Decimal`, use `M`. For example,

```
Double MyVar1 = 3.1415D;
```

.NET Data Types

Note that you can use the standard arithmetic operators such as + (addition), - (subtraction), * (multiplication), / (division), % (modulus), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), == (equal to), != (not equal to), on your numeric variables.

You can also use the increment (++) and decrement (--) unary operators (both before and after the variable). The ~ symbol represents one's complement (thank you…oh, you're welcome!).

The bitwise operators are & (logical AND), | (logical OR), ^ (logical XOR) and ! (logical NOT). You can also use << (bitshift left), >> (bitshift right).

Conditional operators (used in if statements, say) are && (AND), || (OR) and the ternary operator ?:.

Please don't confuse the last two paragraphs! `If` statements are used with &&, ||, and ?:.

Finally, you can perform compound assignment such as `x=x+n` using this form: `x+=n`. Similar for *=, /=, %=, -=, <<=. >>=, &=, |= and ^=.

C# has an operator precedence…don't rely on this…please use enough parentheses to make it clear what you want to happen!

Control Structures

As with many languages, C# has control structures such as if-then-else, while loops, for-loops, etc. We describe these control structures in this section.

The *if* Statement

```
if (expression)
 statement1
else
 statement2
```

```
if (expression) {
 statement1
 statement2
 …
}
```

```
if (expression) {
 statement1
 statement2
 …
}
else {
 statement1
 statemetn2
 …
}
```

```
if (expression1) {
 statement1
 statement2
 …
}
else if (expression2) {
 statement1
 statement2
 …
}
else if (expression3) {
 statement1
 statement2
 …
}
else if (expression4) {
 statement1
 statement2
 …
}
else {
 …
}
```

Note that all of the *expressions* must result in a Boolean value and not simply a numeric value.

Also, note that some authors place the left parentheses ({) on the following line under the "i" in "if". Choose the coding style that fits your lifestyle.

Control Structures

The *switch* Statement

The switch statement works in a similar manner to the if statement. Note that each expression below must be one of the following data types: (s)byte, (u)short, (u)int, (u)long, char, string or an enumerator based on one of the listed types. We talk more about enumerators later on…later…always later…

```
switch (switch_expression)
{
  case expression1:
    statement1
    statement2
    …
    break;
  case expression2:
    statement1
    statement2
    …
    break;
…
 default:
    statement1
    statement2
    …
    break;
}
```

Note that you can combine several case statements together as in:

```
case expression1:
case expression2:
case expression3:
 statement1
 statement2
 …
 break;
```

Control Structures

The *while* Statement

As in other programming languages, the `while` statement will continue to loop while (ha!) the expression evaluates to true.

```
while (boolean-expression)
{
 statement1
 statement2
 …
}
```

Generally, you control the *boolean-expression*'s value from within the loop.

The do/*while* Statement

While the `while` statement only loops when the `boolean-expression` is true, this means that there may be `while` statements that don't do any loops. If you want to have at least one loop performed, then use the `do/while` statement. In other languages, this is similar to how an until statement functions.

```
do
{
 statement1
 statement2
 …
} while (boolean-expression);
```

Control Structures

The *for* Statement

In some cases, you will need to perform a certain number of loops with a counter. For this, you can use the `for` statement:

```
for(initialization-expression; boolean-stopping-expression; iteration-expression)
{
 statement1
 statement2
 …
}
```

For example, let's create

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  for(Int32 i=1; i<11; i++)
  {
   Console.WriteLine(i.ToString());
  }

 }

}
```

Note that you are allowed to nest for statements within each other…oooooh, sounds naughty!!

Control Structures

The *foreach* Statement

Although we only briefly mentioned arrays, there is another control structure that allows access to array (or collection) elements. This is the `foreach` statement.

```
foreach (datatype tmp-var in collection)
{
 statement1
 statement2
 …
}
```

Note that *`datatype`* is one of the listed data types on Slide #12 (as well as others). The *`tmp-var`* is just a temporary variable used within the block of statements. The keyword `in` is required. The *`collection`* variable is the name of the collection.

Here is an example (see next slide). This example creates an array containing three two-letter state codes and then prints them out. Note that we had to add the `System.Collections` namespace in order to obtain access to the `ArrayList()` method. (Just adding `using System;` does not get you **everything** automatically…there's no free lunch, buddy!)

Control Structures

The *foreach* Statement (*continued*)


```
using System;
using System.Collections;

class OurFirstProgram {

 //Main program
 public static void Main() {

  //Create a new empty array.
  ArrayList aStates = new ArrayList();

  //Populate the array with elements.
  aStates.Add("NJ");
  aStates.Add("PA");
  aStates.Add("DE");

  //Print out each state in the array.
  foreach(String sState in aStates)
  {
   Console.WriteLine(sState);
  }

 }

}
```

Control Structures

Two Additional Statements (`break` and `continue`)

You can use the `break` statement within a loop to end the loop at the point `break` appears. This is useful if you want to end the loop if you've reached a certain stopping point.

The `continue` statement stops the execution of the current iteration and sends you back to the top of the loop. This statement is usually used within a `for` statement.

Both of these statements usually appear on a single programming line…alone…solitary…an island…sans amis…:

```
break;

continue;
```

Control Structures

Example

Here is an example that computes the area under the curve x^2+1 from 0 to 10. From calculus, we know that this answer is 343.33…. Let's see how good a rectangular approximation is:

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  //Initialize my variables
  Double SLICE_WIDTH = .0001D; //Width of rectangle
  Double STARTING_X = 0D; //Starting X-Coordinate
  Double ENDING_X = 10D; //Ending X-Coordinate
  Double CURRENT_X = STARTING_X; //Current X value...used for loop
  Double AREA_UNDER_CURVE = 0D; // Area under curve initialized to zero

  //Compute the area under the curve x**2 + 1 from 0 to 10.
  //From calculus, this value is 343.33.
  do
  {
   AREA_UNDER_CURVE += SLICE_WIDTH*(CURRENT_X * CURRENT_X + 1);
   CURRENT_X += SLICE_WIDTH;
  } while(CURRENT_X <= ENDING_X);

  //Print out the results
  Console.WriteLine(AREA_UNDER_CURVE.ToString());

 }

}
```

The answer we receive is 343.33843349694 which is not bad.

Casting Between Types

There may be times that you need to change a value from one type to another data type in order for your program to work properly. For example, you may need to add the integral loop iteration value to a `Single` or `Double` value. To do this, you must explicitly cast to the higher type. To accomplish this, you place the data type within parentheses directly to the left of the value you want to cast. Below, I cast the `Int32` index to a `Double` in order to add:

```
using System;

class OurFirstProgram {

 //Main program
 public static void Main() {

  //Initialize my variable
  Double WIDGET_WIDTH = .123456789D; //Size of widget

  //Loop around 10 times adding the loop index to WIDGET_WIDTH
  for(Int32 indx=1; indx<11; indx++) {

   //Print out the results
   Console.WriteLine( (WIDGET_WIDTH + (Double)indx).ToString());

  }

 }

}
```

You are not limited to casting to numbers of different data types. You can cast to different classes. We will talk about that later on.

Classes

A class is the encapsulation of data and the methods that work on that data. I like to think of a class as a bundle of functionality accessible via a variable. In an object-oriented language, the class does all of the work. Programmatically, classes, fields and methods are defined with certain attributes and modifiers. Your choice of attributes and modifiers – which is a small list of C# keywords – depends on whether you are defining a class, a field, or a method.

The following is the code you use to define a *class*:

```
attributes modifiers class name-of-your-class [:baseClassName]
{
  fields and methods
}
```

where attributes and modifiers can be:

1. attributes -
   a. `abstract` – indicates that no instances of the type can be constructed
   b. `sealed` – indicates that the type cannot be used as a base class
   c. `static` – indicates that the class contains only static members. You cannot instantiate this type of class
   d. `unsafe` – denotes an unsafe context which is required for any operation involving pointers
   e. `partial` – indicates that the definition of the class itself is split among several class definitions over two or more source files

Classes

2. modifiers:
   a. `public` – indicates that this class is publically available
   b. `internal` – (the default) indicates that this class is available only from within the defining assembly. We talk more about assemblies later on.

For *methods* within a class, the attributes and modifiers can be slightly different. Here is how to define a *method* within a class:

```
attributes modifiers name-of-your-method(parameters)
{
  statements
  return-value-if-necessary
}
```

where attributes and modifiers can be:

1. attributes:
   1. `abstract` – indicates that the derived class must override this method
   2. `virtual` – indicates that this method can be overridden by a derived class
   3. `override` – indicates that the derived method is overriding the base method
   4. `sealed` – indicates that this method cannot be overridden by a derived type
   5. `new` – indicates that this method has nothing to do with a similarly named method that may appear in the base class

Classes

2. modifiers:
   a. `public` – indicates that this method is publically accessible
   b. `internal` – indicates that this method is accessible only from within the defining assembly.
   c. `private` – indicates that this method only accessible from within the class itself and not outside
   d. `protected` – indicates that this method is accessible from any derived class

In addition to those listed above, a method can be labeled `static` indicating that it can be executed without instantiation. As we saw, our Main method is `static` as well as `public`. Also, methods can be labeled as `unsafe` as well as `extern`.

For *fields* (properties, attributes, variables) within a class or method, they can be labeled as `static`, `public`, `internal`, `private` (default), `protected`, `new`, `unsafe`, `readonly`, and `volatile`. The modifier `readonly` indicates that the field cannot be changed after it has been initialized.

Finally, a field can be marked as a constant using the `const` keyword. A constant's value can never change. During the compilation of your program, every mention of a constant throughout is substituted with your constant value.

Note that, for fields, along with the above list, you are still required to enter a datatype as well as a field name. For example,

```
public const String msg = "Here is my message!";
```

Classes

Finally, we saw the `new` keyword above related to methods. In that context, `new` indicates that the method is not related to a base class method. The `new` keyword also is used to instantiate an object from a class. You won't confuse the two…`new` is mostly used to instantiate an object. We explain this later on…

Let's redo our area under the curve example by creating a class specifically designed to compute the area under $x^2 + 1$. Then, let's instantiate this class in the Main program, execute the method to do the computation and get some results back. (Code is continued on next slide.)

```
using System;

1  class AreaUnderCurve {

2   //Initialize my variables
   public Double SLICE_WIDTH = .0001D; //Width of rectangle
   public Double STARTING_X = 0D; //Starting X-Coordinate
   public Double ENDING_X = 10D; //Ending X-Coordinate
   public Double CURRENT_X = 0D; //Current X value...used for loop
   public Double AREA_UNDER_CURVE = 0D; // Area under curve initialized to zero

   //Create a method to compute the area
3  public Double ComputeArea() {

    do
    {
     AREA_UNDER_CURVE += SLICE_WIDTH*(CURRENT_X * CURRENT_X + 1);
     CURRENT_X += SLICE_WIDTH;
    } while(CURRENT_X <= ENDING_X);

    //Return the value to the caller
4   return(AREA_UNDER_CURVE);

   }

  }
```

Classes


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create a variable to hold the resulting area
  Double area = 0;

  //Using the new keyword, instantiate the class
  AreaUnderCurve auc = new AreaUnderCurve();

  //Call the ComputeArea method to compute the area.
  area = auc.ComputeArea();

  //Print out the results
  Console.WriteLine(area.ToString());

 }

}
```

The previous slide defines the class `AreaUnderCurve`. Here is a description of the code for this class:

1. This line defines the class as `AreaUnderCurve`. Note that we did not use the `public` keyword since the default of `internal` is fine.
2. These five lines create public variables and initializes them. Note that the line defining CURRENT_X has STARTING_X replaced with a 0D. That is because you cannot reference a non-static variable at this point in the code.
3. This line defines our method `ComputeArea`. Note that this method does not take any parameters. We define the method as public so we can run it from `Main`.
4. This line is responsible for returning the computed value back to the main program. Note that we define the return type for the method to be `Double`.

Classes

The previous slide defines the class `MainProgram`. Here is a description of the code for this class:

5. This line creates and initializes a variable to hold the resulting area. Since the method `ComputeArea` returns a `Double`, this variable must be a `Double`.
6. This line is responsible for instantiating our class `AreaUnderCurve`. Note that the class name appears to the left of the variable name, `auc`, and also after the `new` keyword. The mention after the `new` keyword allows you to pass values into the class during instantiation. We talk more about that later on.
7. This line executes the method `ComputeArea`. Note how this is coded: the instantiated variable is followed by a period and then the method name. Note, also, that lines 5 and 7 can be combined into a single line.
8. Finally, we write out the results using the `ToString` method. Remember that all classes are derived from the base class `System.Object` which has `ToString` – and you will recall – several other methods defined for it.

Note that we've created the variables `SLICE_WIDTH`, etc. as `public` variables. This will allow us to change these variables via the `auc` variable. For example, `auc.SLICE_WIDTH=.00001D`. This may be what you want to allow, but more than likely you want some way to completely prevent anyone from changing these variables at all; or, you want to allow for changes, but to be able to check if the value makes sense.

To completely prevent anyone from changing these variables via the `auc` variable, change our six variables from `public` to `private`. This gives the methods within the class access to them, but not from the `auc` variable in the Main program.

Classes

Now, if you don't want to be totally Shakerian and prevent complete access to your variables (via `private`), but you also don't want to be a slut and allow spread-eagle access to your variables (via `public`), you have three more dignified ways available to you:

1. Create a constructor for your class. A *constructor* is a method with the same name as your class and is executed when the `new` keyword is used to instantiate your class. This allows you to initialize certain variables in the object instantiated from your class. A constructor method does not return a value (recall that our `ComputeArea` method returns a `Double`), but it can accept one or more values as parameters.
2. Despite the use of the word variable up to this point in the presentation, the words field and attribute are probably more standard (as well as *member variables*). Now, you can allow read-from and write-to access to the properties in your class via *getters* and *setters* which are just functions that allow you ..uh… read-from and write-to access to the properties in your class. That is, you can create additional methods such as `get_SLICEWIDTH()` and `set_SLICEWIDTH(value)` in order to change your properties. Alas, this approach to altering properties is violently discouraged. You should use smart fields instead…as described in #3 below.
3. Smart fields are sanctioned getters and setters as described in #2 above. Smart fields are also referred to as *properties*.

We describe constructors and smart fields next. Since #2 is not sanctioned – #2 stinks, even – we will not describe this approach.

Constructors

In order to create a constructor, you create a public method within your class with the same name as your class. The constructor is called when the `new` keyword is used to instantiate your class. Below, we change our `AreaUnderCurve` class to use a constructor:

```
using System;

class AreaUnderCurve {

 //Initialize my variables
 private Double SLICE_WIDTH; //Width of rectangle
 private Double STARTING_X; //Starting X-Coordinate
 private Double ENDING_X; //Ending X-Coordinate
 private Double CURRENT_X; //Current X value...used for loop
 private Double AREA_UNDER_CURVE; // Area under curve initialized to zero

 //Create a constructor for this class to initialize our values.
 //This constructor is used when no parameters have been passed.
 public AreaUnderCurve() {

  SLICE_WIDTH = .0001D;
  STARTING_X = 0D;
  ENDING_X = 10D;
  CURRENT_X = STARTING_X;
  AREA_UNDER_CURVE = 0D;

 }

 //This constructor is used when parameters are passed in.
 public AreaUnderCurve(Double pSLICEWIDTH,Double pSTARTINGX,Double pENDINGX) {

  SLICE_WIDTH = pSLICEWIDTH;
  STARTING_X = pSTARTINGX;
  ENDING_X = pENDINGX;
  CURRENT_X = STARTING_X;
  AREA_UNDER_CURVE = 0D;

 }
```

1

2

Constructors


```
//Create a method to compute the area
public Double ComputeArea() {

 do
 {
  AREA_UNDER_CURVE += SLICE_WIDTH*(CURRENT_X * CURRENT_X + 1);
  CURRENT_X += SLICE_WIDTH;
 } while(CURRENT_X <= ENDING_X);

 //Return the value to the caller
 return(AREA_UNDER_CURVE);

 }

}
```

As you see on the previous slide, I have thrown you for a loop. I created two constructors. The first constructor has no parameters and sets all of the properties itself. The second constructor has three parameters and they are used within the code of that constructor to allow you to initialize certain properties. Now, you might think that the C# compiler would be confused that there are two methods with the same name defined. In fact, C# sees them as different because one has parameters whereas the other has no parameters. As you see in the code below, I changed my instantiation of `auc` to include the parameter values I want to pass, for example:

```
//Using the new keyword, instantiate the class
AreaUnderCurve auc = new AreaUnderCurve(.00001,0,10);
```

Smart Fields

Another way to allow access to your class's properties is through *smart fields*. These are fields that have a getter and setter defined for them. The syntax for smart fields is as follows:

```
attributes modifiers datatype property_name
{

  set {
   …code to set property value goes here…
  }

  get {
   …code to retrieve property value goes here…
  }

}
```

Now, the attributes and modifiers are the usual suspects: `static`, `public`, `internal`, `protected`, `private`, `new`, `virtual`, `abstract`, `override`, `sealed`, `unsafe` and `extern`. For the most part, though, your smart fields will be `public`. Note that if you have both a getter and a setter, the property is called *read-write*. If you have only a getter, the property is *read-only*. If you have only a setter, the property is *write-only*. If you have neither, you say nothing.

In the books I've read, the property_name seems to be just a camel-cased version of the property you are trying to get/set. For example, if your property is called x, then the property_name would be X. But, you can choose your own coding convention…knock yourself out.

Smart Fields

Now, let's modify our code to use smart fields. Note that you do not have to eliminate the constructors; smart fields can be used in addition to constructors. Note that the use of the words *get* and *set* are considered a no-no in smart field world and should not appear in the property_name…sorry, kids!

```
using System;

class AreaUnderCurve {

 //Initialize my variables
 private Double SLICE_WIDTH; //Width of rectangle
 private Double STARTING_X; //Starting X-Coordinate
 private Double ENDING_X; //Ending X-Coordinate
 private Double CURRENT_X; //Current X value...used for loop
 private Double AREA_UNDER_CURVE; // Area under curve initialized to zero

 //Create a constructor for this class to initialize our values.
 //This constructor is used when no parameters have been passed.
 public AreaUnderCurve() {

  SLICE_WIDTH = .0001D;
  STARTING_X = 0D;
  ENDING_X = 10D;
  CURRENT_X = STARTING_X;
  AREA_UNDER_CURVE = 0D;

 }

 //Define smart fields
 public Double SliceWidth {
  set {
   SLICE_WIDTH=value;  //value contains the value
  }

  get {
   return SLICE_WIDTH;
  }
 }

 public Double StartX {
  set {
   STARTING_X=value;  //value contains the value
  }

  get {
   return STARTING_X;
  }
 }
```

Smart Fields


```
public Double EndX {
 set {
  ENDING_X=value;  //value contains the value
 }

 get {
  return ENDING_X;
 }
}

//Create a method to compute the area
public Double ComputeArea() {

 do
 {
  AREA_UNDER_CURVE += SLICE_WIDTH*(CURRENT_X * CURRENT_X + 1);
  CURRENT_X += SLICE_WIDTH;
 } while(CURRENT_X <= ENDING_X);

 //Return the value to the caller
 return(AREA_UNDER_CURVE);

}

}
```

As you see on this slide as well as the previous slide, I have created three smart fields: `SliceWidth`, `StartX` and `EndX`. Note the use of the keyword `value`. This is the actual value being passed into the setter. Also, note the use of the keyword `return`. This is used in the getter to return the value back to the thing doing the get-ting. Within the getters and setters you can write code to check for boo-boo's. On the next slide, we show you how to use the getters and setters. After instantiating our class, but before calling `ComputeArea`, we use the setters to set the values to our desired amounts. Then, we call `ComputeArea`. Next, we use the three getters to display the values we actually used…nice to know that, huh?

Smart Fields


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create a variable to hold the resulting area
  Double area = 0;

  //Using the new keyword, instantiate the class
  AreaUnderCurve auc = new AreaUnderCurve();

  //Before calling ComputeArea, use Smart Fields to
  //set values to desired quantities.
  auc.SliceWidth = .0000001D;
  auc.StartX = 0D;
  auc.EndX = 10D;

  //Call the ComputeArea method to compute the area.
  area = auc.ComputeArea();

  //Print out the results
  Console.WriteLine(auc.SliceWidth.ToString());
  Console.WriteLine(auc.StartX.ToString());
  Console.WriteLine(auc.EndX.ToString());
  Console.WriteLine(area.ToString());

 }

}
```

In the `Main` program above, the code in red shows you how to call the setters and the code in purple shows you how to call the getters. Sweet, huh? This is much more colorful than using methods!

Derived Classes

As mentioned in the first presentation about object-oriented programming, you can create a class, say CLASS1, and then create a second class, say CLASS2, that uses CLASS1's fields and methods. That is, CLASS1 is the *base class* and CLASS2 is the *derived class*. And this is *good* because the derived class has access (for the most part) to all of the base class's fields and methods.

As you can imagine, this can present its own problems…oy! it presents problems like you wouldn't believe!… For example, what happens if a method in the base class is also the same name as a method in the derived class? This shouldn't happen if you've created both, but what if you got your base class from a shady dealer on the mean streets of the big city?

Let's leave that for the moment and concentrate on working with derived classes. See next four slides for code.

Derived Classes


```
using System;

//Create a base class to contain several methods to
//approximate the area under the integral x**2 + 1.
class ApproxMethod {

 private Double SLICE_WIDTH;
 private Double STARTING_X; //Starting X-Coordinate
 private Double ENDING_X; //Ending X-Coordinate

 //Define smart fields
 public Double SliceWidth {
  set {
   SLICE_WIDTH=value;  //value contains the value
  }

  get {
   return SLICE_WIDTH;
  }
 }

 public Double StartX {
  set {
   STARTING_X=value;  //value contains the value
  }

  get {
   return STARTING_X;
  }
 }

 public Double EndX {
  set {
   ENDING_X=value;  //value contains the value
  }

  get {
   return ENDING_X;
  }
 }
```

Derived Classes


```
//Define a private method which returns x**2 + 1
 private Double F(Double pX) {

  return( (Double)(pX*pX + 1));

 }

 //Rectangle Rule
 protected Double RectangleRule() {

              Double CURRENT_X = STARTING_X;
              Double AREA_UNDER_CURVE = 0D;

  do
  {
   AREA_UNDER_CURVE += SLICE_WIDTH*F(CURRENT_X);
   CURRENT_X += SLICE_WIDTH;
  } while(CURRENT_X <= ENDING_X);

  return(AREA_UNDER_CURVE);

 }

 //Trapezoidal Rule
 protected Double TrapezoidalRule() {

              Double CURRENT_X = STARTING_X;
              Double AREA_UNDER_CURVE = 0D;
  Double F_CURRENT_SLICE; //height at current slice
  Double F_NEXT_SLICE; //height at next slide

  do
  {

   F_CURRENT_SLICE = F(CURRENT_X);
   CURRENT_X += SLICE_WIDTH;
   F_NEXT_SLICE = F(CURRENT_X);

   if (CURRENT_X <= ENDING_X) {
    AREA_UNDER_CURVE += SLICE_WIDTH*(F_CURRENT_SLICE + F_NEXT_SLICE)/2;
   }

  } while(CURRENT_X <= ENDING_X);

  return(AREA_UNDER_CURVE);

 }
```

Derived Classes


```
//Simpson's Quadratic Approximation Rule
 protected Double SimpsonsRule() {

  Double B_MINUS_A_DIV_6;
  Double F_A;
  Double F_A_PLUS_B_DIV_2;
  Double F_B;

  B_MINUS_A_DIV_6 = (ENDING_X - STARTING_X) / 6;
  F_A = F(STARTING_X);
  F_A_PLUS_B_DIV_2 = F( (STARTING_X + ENDING_X) / 2 );
  F_B = F(ENDING_X);

  return( (Double)( B_MINUS_A_DIV_6*( F_A + 4*F_A_PLUS_B_DIV_2 + F_B) ) );

 }

}

//Create the derived class AreaUnderCurve
class AreaUnderCurve : ApproxMethod {

 private Double AREA_UNDER_CURVE;

 public Double ComputeArea(Int32 pWhich) {

  if (pWhich == 1) {
   AREA_UNDER_CURVE =  RectangleRule();
   Console.WriteLine("Rectangle Rule");
  }
  else if (pWhich == 2) {
   AREA_UNDER_CURVE =  TrapezoidalRule();
   Console.WriteLine("Trapezoidal Rule");
  }
  else if (pWhich == 3) {
   AREA_UNDER_CURVE =  SimpsonsRule();
   Console.WriteLine("Simpson's Rule");
  }
  else {
   AREA_UNDER_CURVE =  RectangleRule();
   Console.WriteLine("Rectangle Rule");
  }

  return(AREA_UNDER_CURVE);

 }

}
```

Derived Classes


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create a variable to hold the resulting area
  Double areaRR = 0;
  Double areaTR = 0;
  Double areaSR = 0;

  //Using the new keyword, instantiate the class
  AreaUnderCurve auc = new AreaUnderCurve();

  //Use the smart fields to initialize the values
  auc.SliceWidth = .0001D;
  auc.StartX = 0D;
  auc.EndX = 10D;

  //Call the ComputeArea method to compute the area.
  areaRR = auc.ComputeArea(1);
  Console.WriteLine(areaRR.ToString());
  areaTR = auc.ComputeArea(2);
  Console.WriteLine(areaTR.ToString());
  areaSR = auc.ComputeArea(3);
  Console.WriteLine(areaSR.ToString());

 }

}
```

As you can see on the previous slide, the code in red indicates that we have created our class `AreaUnderCurve` by *deriving* it from the base class `ApproxMethod`: **`class AreaUnderCurve : ApproxMethod {`**

By doing this, we can access all of the public and protected fields and methods, but also have the chance to create a method of our own, `ComputeArea` in the base class.

Now, the method named `SimpsonsRule` uses the Quadratic Approximation formula. What happens if we want to override this and use another one of Simpson's wonderful approximation formulas?

Derived Classes

In this case, we can create a method *with the same name*, `SimpsonsRule`, and place it in our own class. In this case, you must declare the replacement method with the keyword `new`. Below is our new `SimpsonsRule` within the `AreaUnderCurve` class:

```
//Use Simpson's 3/8 Approximation Rule instead!!
new protected Double SimpsonsRule() {

 Double B_MINUS_A_DIV_8;
 Double F_A;
 Double F_2A_PLUS_B_DIV_3;
 Double F_A_PLUS_2B_DIV_3;
 Double F_B;

 B_MINUS_A_DIV_8 = (ENDING_X - STARTING_X) / 8;
 F_A = F(STARTING_X);
 F_2A_PLUS_B_DIV_3 = F( (2*STARTING_X + ENDING_X) / 3 );
 F_A_PLUS_2B_DIV_3 = F( (STARTING_X + 2*ENDING_X) / 3 );
 F_B = F(ENDING_X);

 return( (Double)( B_MINUS_A_DIV_8*( F_A + 3*F_2A_PLUS_B_DIV_3 + 3*F_A_PLUS_2B_DIV_3 + F_B) ) );

}
```

Now, if you attempt to compile this you will receive several **errors** due to the fact that the code above is using the fields `ENDING_X`, `STARTING_X` as well as our helper method `F()`. Since these are all defined as `private`, change them to `protected` and the new code will compile.

Derived Classes

As mentioned earlier, derived classes present some problems. One problem is what happens if there is a constructor in the base class as well as in the derived class. Are both constructors called or just one? If both, which constructor is called first? Can you control which constructor is called at what time? Can you feel a migraine coming on?

There are two *constructor initializers* which will help answer these questions. The two constructor initializers are:

1. `base(…)` – Allows you to call the current class's *base class* constructor.
2. `this(…)` – Allows you to call a constructor *within the current class*. This is good if you have overloaded several constructors (that is, you have constructors with different sets of parameters).

First, at a very simple level, let's see the order of constructor calls. Here is a simple example (see next slide).

You will find that the Base class's constructor is called *first* followed by the derived class's constructor. This is just what you probably thought!

Derived Classes


```
using System;

class MyBaseClass {

 public MyBaseClass() {
  Console.WriteLine("In the Base Class constructor!!");
 }

}

class MyDerivedClass : MyBaseClass {

 public MyDerivedClass() {
  Console.WriteLine("In the Derived Class constructor!!");
 }

}

class MainProgram {

 //Main program
 public static void Main() {

  MyDerivedClass mdc = new MyDerivedClass();

 }


}
```

Derived Classes

Now, let's add `base()` to our derived class's constructor call and recompile and run the code:

```
public MyDerivedClass() : base() {
 Console.WriteLine("In the Derived Class constructor!!");
}
```

Lo and behold, the results are exactly the same! The power comes in when there are several base class constructors each with a different set of parameters. You can then use `base()` to call the exact constructor you want based on the parameters you supply to the `base()` keyword. Here is our base class:

```
class MyBaseClass {

 public MyBaseClass() {
  Console.WriteLine("In the Base Class constructor!!");
 }

 public MyBaseClass(Int32 pParm1) {
  Console.WriteLine("In the Base Class constructor: Int32 pParm1");
 }

}
```

Here is our derived class's constructor call now:

```
public MyDerivedClass() : base(5) {
 Console.WriteLine("In the Derived Class constructor!!");
}
```

As you see, we are passing the integer 5 as a parameter to the `base()` method.

Derived Classes

In this case, the results are as follows:

```
In the Base Class constructor: Int32 pParm1
In the Derived Class constructor!!
```

Note that the base class constructor with the single parameter was executed, but the base class constructor with no parameters was *not* called! Finally, the derived class constructor was called.

As you can see, this gives you more control over which base class constructor is actually called from the derived class.

Now, as you know, if you have several constructors within your derived class, all of which have different sets of parameters, the constructor that is called is based on the number and type of parameters passed during the instantiation of the class (using the `new` keyword, say, from the `Main` program). Now, you can tell any one of these constructors to call one of the other constructors within the same class. Thus, you can set up one constructor to do most of the work while the other constructors do some of the work. You do this by using the `this()` keyword. Here is an example (see next slide):

Derived Classes


```
using System;

class MyClass  {

 private Int32 iVar1;
 private Int32 iVar2;

 //Initialize all fields to default values
 public MyClass() {
  Console.WriteLine("In the parameterless constructor!!");
  iVar1 = -1;
  iVar2 = -1;
 }

 //Update iVar1 based on pParm1...note that the
 //parameterless constructor is called first!!
 public MyClass(Int32 pParm1) : this() {
  Console.WriteLine("In the constructor with one parameter!!");
  iVar1 = pParm1;
 }

 //Update iVar1 and iVar2 based on pParm1 and pParm2...note that the
 //parameterless constructor is called first!!
 public MyClass(Int32 pParm1,Int32 pParm2) : this() {
  Console.WriteLine("In the constructor with two parameter!!");
  iVar1 = pParm1;
  iVar2 = pParm2;
 }

}
```

…continued on next slide…

Derived Classes


```
class MainProgram {

 //Main program
 public static void Main() {

  MyClass mc1 = new MyClass();
  Console.WriteLine();

  MyClass mc2 = new MyClass(5);
  Console.WriteLine();

  MyClass mc3 = new MyClass(5,6);
  Console.WriteLine();

 }


}
```

In this case, the results are as follows.

```
In the parameterless constructor!!

In the parameterless constructor!!
In the constructor with one parameter!!

In the parameterless constructor!!
In the constructor with two parameter!!
```

As you see, the parameterless constructor is ALWAYS called first when using `this()`…which means that you must have a parameterless constructor for this to work!!

Derived Classes

Now, the example on the previous slide did not take into account any base class, but was a standalone class. Let's see the order of execution when inheritance is involved.

```
using System;

class MyBaseClass {
 public MyBaseClass() {
  Console.WriteLine("Inside parameterless BASE CLASS constructor!");
 }
}

class MyDerivedClass : MyBaseClass {

 private Int32 iVar1;
 private Int32 iVar2;

 //Initialize all fields to default values
 public MyDerivedClass() {
  Console.WriteLine("In the parameterless constructor!!");
  iVar1 = -1;
  iVar2 = -1;
 }

 //Update iVar1 based on pParm1...note that the
 //parameterless constructor is called first!!
 public MyDerivedClass(Int32 pParm1) : this() {
  Console.WriteLine("In the constructor with one parameter!!");
  iVar1 = pParm1;
 }

 …continued on next slide…
```

Derived Classes


```
 //Update iVar1 and iVar2 based on pParm1 and pParm2...note that the
 //parameterless constructor is called first!!
 public MyDerivedClass(Int32 pParm1,Int32 pParm2) : this() {
  Console.WriteLine("In the constructor with two parameter!!");
  iVar1 = pParm1;
  iVar2 = pParm2;
 }

}

class MainProgram {

 //Main program
 public static void Main() {

  MyDerivedClass mc1 = new MyDerivedClass();
  Console.WriteLine();

  MyDerivedClass mc2 = new MyDerivedClass(5);
  Console.WriteLine();

  MyDerivedClass mc3 = new MyDerivedClass(5,6);
  Console.WriteLine();

 }

}
```

The results are as follows (see next slide):

Derived Classes

```
Inside parameterless BASE CLASS constructor!
In the parameterless constructor!!

Inside parameterless BASE CLASS constructor!
In the parameterless constructor!!
In the constructor with one parameter!!

Inside parameterless BASE CLASS constructor!
In the parameterless constructor!!
In the constructor with two parameter!!
```

As you can see, the BASE CLASS constructor is always called first, followed by the parameterless derived class constructor and then the more specific derived class constructor.

Static Classes

In the `System` namespace, there is a class called `Math`. This class contains several public mathematical methods, some of which are:

1. `Math.Sqrt(x)` – returns the square root of x
2. `Math.Pow(x,y)` – returns the number x raised to the power y
3. `Math.Abs(x)` – returns the absolute value of x

…and so on…

There are also two public fields defined in this class:

1. `Math.E` – returns the value 2.7182818284590452354
2. `Math.PI` – returns the value 3.14159265358979323846

Note that all of these methods as well as the two fields are *static*. That is, there is *no need to instantiate* the class in order to use the methods or fields in your code. As you can imagine, this saves a lot of time – much like Tribbles being born pregnant – having to instantiate this class every time you just want to take an absolute value of a number or get the value of pi.

For example, here is how you would use the static method `Math.Abs()`:

```
Double aval1;
aval1 = Math.Abs(-75.6 * Math.PI);
```

You can find more on the `Math` class on the following MSDN web page:

http://msdn.microsoft.com/en-us/library/system.math.aspx

Static Classes

Now, you can create your own static class if you want by using the `static` keyword when you create your class. When you do so, all of the members (fields, methods, etc.) must be static. Here is a simple example,

```
using System;

public static class MyMath {

 //Euler-Mascheroni Constant
 public static Double EulerMascheroni = 0.57721566490153286060651209008240243104215933593992D;

 //Gelfond's Constant
 public static Double GelfondConstant = Math.Pow(Math.E,Math.PI);

 //Champernowne's Cconstant
 public static Double ChampernowneConstant = 0.12345678910111213141516D;

 //Catalan's Constant
 public static Double CatalanConstant = 0.91596559417721901505460351493238411077 4D;

 //Compute the distance between two points
 public static Double DistanceBetweenTwoPoints(Double X1,Double Y1, Double X2, Double Y2) {
  return(Math.Sqrt( Math.Pow(X2-X1,2) + Math.Pow(Y2-Y1,2) ));
 }

}

class MainProgram {

 //Main program
 public static void Main() {

  Console.WriteLine(MyMath.CatalanConstant.ToString());
  Console.WriteLine(MyMath.DistanceBetweenTwoPoints(0,0,1,1).ToString());

 }

}
```

Static Classes

Note that while we defined our constants as `public static`, Microsoft seems to use `public const` for their static fields (at least for `Math.PI` and `Math.E`). Constant fields are static in nature, so it probably doesn't matter which you use except, you'll recall, that constants are replaced (physically, by a man named Bob) throughout your code during compile time.

Recall that in the last section we talked briefly about classes. Now, you are allowed to have a static field within a non-static class with the benefit that no matter how many instances of your class you create, *the static field is shared among all of the instances*. Normally, when you instantiate a class, all of the fields belong to that particular instance. Not so when the field is marked as static. Here is an example (which is continued on the next slide):

```
using System;

class MyClass {

 //Track how many instances have been created!
 public static Int32 InstanceCount = 0;

 //Update the static member InstanceCount
 public MyClass() {

  InstanceCount++;

 }

 //Create a smart field to return the InstanceCount
 public Int32 InstCnt {
  get {
   return(InstanceCount);
  }
 }

}
```

Static Classes


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create the first instance
  MyClass cls1 = new MyClass();

  //Display the instance count using cls1
  Console.WriteLine(cls1.InstCnt.ToString());

  //Create the second instance
  MyClass cls2 = new MyClass();

  //Display the instance count using cls2
  Console.WriteLine(cls2.InstCnt.ToString());

  //Create the third instance
  MyClass cls3 = new MyClass();

  //Display the instance count using cls3
  Console.WriteLine(cls3.InstCnt.ToString());

  //Circle back to cls1 and see what the InstanceCount is!
  Console.WriteLine(cls1.InstCnt.ToString());

 }

}
```

As you would suspect, the results are:

```
1
2
3
3
```

Operator Overloading

With all of these object thingies floating around, you may think that it may get complicated working with two or more objects. For example, given two `Point` objects, defined as taking an X and Y coordinate, you may ask how you would "add" these two points together in C# and then return a new point object. There are two ways to do this:

1. Create a method that takes an object as a parameter and then add it to the calling object. For example: `p1.Add(p2)` and return the appropriate object.
2. Overload the "+" sign (called an *operator*) to accept two `Point` objects and return the sum of the coordinates in a `Point` object. This is called *operator overloading*.

We show both methods in this section. First, bullet point #1 starts on the next slide:

Operator Overloading


```
using System;

1  class PointClass {

2   //(X,Y)-Coordinate initialized to (0,0)
    public Double X = 0;
    public Double Y = 0;

3   //Initialize the Point
    public PointClass(Double pX,Double pY) {

     this.X = pX;
     this.Y = pY;

    }

    //Create a method to add two Points together
4   public PointClass Add(PointClass pPoint1) {

5    PointClass pPointAdded = new PointClass(0,0);

     pPointAdded.X = pPoint1.X + this.X;
     pPointAdded.Y = pPoint1.Y + this.Y;

     return(pPointAdded);

    }

}
```

…continued on next slide…

Operator Overloading


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create the first point
  PointClass P = new PointClass(5,10);

  //Create the second point
  PointClass Q = new PointClass(50,100);

  //Create a point to hold all the summed point
  PointClass S = new PointClass(0,0);

  //Add the points together
  S = P.Add(Q);

  //Write out the X and Y Coordinate
  Console.WriteLine(S.X.ToString());
  Console.WriteLine(S.Y.ToString());

 }

}
```

1. We create our class called `PointClass`. class which has two public fields, X and Y, which hold the coordinate.
2. Our class has two public fields, X and Y, which hold the coordinate.
3. The sole constructor sets the point to the appropriate values.

Operator Overloading

4. We create the `Add` method which takes a `PointClass` object as a parameter and then adds it to the point which called the class. The point which called the method is an object of type `PointClass` itself, so the `this` keyword helps to distinguish which X and Y coordinates you are talking about. Since the parameter we are passing in is called `pPoint1`, the X-Coordinate is referred to as `pPoint1.X` and the Y-Coordinate is referred to as `pPoint.Y`. The `this.X` and `this.Y` are the X- and Y-Coordinates of the calling `PointClass` object.
5. Also, note that in order to return a `PointClass` object, we create a new `PointClass` object and fill in its X- and Y-Coordinate values by adding the `this.X` to `pPoint1.X`, and `this.Y` to `pPoint1.Y`. Finally, we return the `PointClass` object.
6. In the Main program, we create three `PointClass` objects, P, Q and S, and then add two of them together, P and Q, and receive the `PointClass` object from the Add method into the S object.

Next, let's use operator overloading to overload the plus-sign to replace the `Add` method. All of the code stays the same, except we add the following method to the `PointClass` class:

```
//Create an overload for the plus-sign (+)
public static PointClass operator+(PointClass pPoint1,PointClass pPoint2) {

 PointClass pPointAdded = new PointClass(0,0);

 pPointAdded.X = pPoint1.X + pPoint2.X;
 pPointAdded.Y = pPoint1.Y + pPoint2.Y;

 return(pPointAdded);

}
```

Operator Overloading

Now, here is an example of how to call the plus-sign operator within the Main program:

```
//Create a point to hold all the summed point
PointClass T = new PointClass(0,0);

//Add the points together
T = P+Q;
```

On the previous slide, the plus-sign is overloaded using the `operator` keyword followed by the plus-sign symbol (+). The parameters for this method, unlike our `Add` method, take two `PointClass` objects as parameters. As you see above, the first parameter referred to P and the second parameter refers to Q.

Note that operator overloading requires that the method be `static`.

Also, you cannot overload all of the operators, just some of them.

Structures Are Not Dead!

For those of you who know some C or C++, you may well wonder if C# has structures (or `struct`s in geekspeak) built-in? Of course it does, silly-billy! As a matter of fact, `struct`s and `class`es are very similar except for these trifling points:

1. If your structure has a constructor (I'm a poet!), then said constructor must have at least one parameter. Unlike classes, structures are not allowed to have parameterless constructors.
2. Structures cannot have finalizers (we'll talk more about these later on), but classes can.
3. Structures cannot have virtual fields and methods whereas classes can.
4. Structures are *value types*, classes are *reference types*. Recall that value types are stored on the thread's stack whereas reference types are stored on the big heaping heap.
5. Structures do not support inheritance, classes most certainly do!

So, recall the code from the last section. I used a class called `PointClass` to initialize and add points together. Now, each point required that a class be instantiated. Another approach – and some would say better – is to create a `Point` structure to hold only the (x,y)-coordinate and leave the `PointClass` to add them together. Let's see this inaction…I mean, *in action*:

Structures Are Not Dead!


```
using System;

struct Point {
 public Double X;
 public Double Y;

 public Point(Double pX,Double pY) {
  X = pX;
  Y = pY;
 }

 //Create a method to add two Points together
 public Point Add(Point pPoint1) {

  Point pPointAdded = new Point(0,0);

  pPointAdded.X = pPoint1.X + this.X;
  pPointAdded.Y = pPoint1.Y + this.Y;

  return(pPointAdded);

 }

}
```

…continued on next slide…

Structures Are Not Dead!


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create the first point
  Point P = new Point(5,10);

  //Create the second point
  Point Q = new Point(50,100);

  //Create a point to hold all the summed point
  Point S = new Point(0,0);

  //Add the points together
  S = P.Add(Q);

  //Write out the X and Y Coordinate
  Console.WriteLine(S.X.ToString());
  Console.WriteLine(S.Y.ToString());

 }

}
```

As you see, we are *solely* using a structure and no classes in this example. The next example uses a combination of structures and classes.

Structures Are Not Dead!


```
using System;

struct Point {

 public Double X;
 public Double Y;

}

class PointClass {

 //Create a method to add two Points together
 public Point Add(Point pPoint1,Point pPoint2) {

  Point pPointAdded = new Point();

  pPointAdded.X = pPoint1.X + pPoint2.X;
  pPointAdded.Y = pPoint1.Y + pPoint2.Y;

  return(pPointAdded);

 }

}
```

…continued on next slide…

Structures Are Not Dead!


```
class MainProgram {

 //Main program
 public static void Main() {

  //Create the first point
  Point P = new Point();
  P.X=5;
  P.Y=10;

  //Create the second point
  Point Q = new Point();
  Q.X=50;
  Q.Y=100;

  //Create a point to hold all the summed point
  Point S = new Point();

  //Create an instantiation of the class.
  PointClass pc = new PointClass();

  //Add the points together
  S = pc.Add(P,Q);

  //Write out the X and Y Coordinate
  Console.WriteLine(S.X.ToString());
  Console.WriteLine(S.Y.ToString());

 }

}
```

Structures Are Not Dead!

As you can see, we are using a structure to hold the point coordinates, while the class is responsible for doing the work of adding the two points together.

Now, of these three examples, I don't know which is "right". That is, is it better to create all of the data and code within a structure? Or is it better to create the data and code solely within a class? Or is a combination of the two better?

My guess would be that a combination of the two is correct. That is, create a structure to hold the data (points, in this case), and create a class to work off of those structures. The reason I am suggesting this is because structures are *value types* and are held on the thread's stack rather than off the heap and access to a structure's data is probably faster due to that fact. The addition of two structures is probably faster than the addition of two classes held on the heap. Let me know what you think.

Arrays

Creating variables to hold numbers and strings is nice, but eventually you will want to create an array to hold many numbers or strings or any objects you want. This section describes how to create and manipulate arrays.

To declare an array, you place empty square brackets to the right of the data type where it occurs:

```
datatype[] array-name = new datatype[#];
```

where datatype is the desired data type (such as `Double`, `Int32`, etc.)

array-name is the name of your array

is the number of elements in your array.

For example, let's declare an array to hold 4 `Double`s:

```
Double[] aMyValues = new Double[4];
```

Now, the array `aMyValues` has four slots available for you to use an each slot can contain a `Double`.

As in may high-level, fancy-shmancy programming languages, arrays are zero-based, meaning that our array's indexes range from 0 to 3, and NOT 1 to 4. Yes, it's a pain in the ass, but I'm just a cog in the wheel of life!

Arrays

Now, let's assign values to the array:

```
aMyValues[0] = 1.0000D;
aMyValues[1] = 1.4142D;
aMyValues[2] = 1.7321D;
aMyValues[3] = 2.0000D;
```

Now, you'll notice that these are just the square roots of the numbers from 1 to 4. Let's use a for-loop to compute the same thing:

```
for(Int32 indx=0; indx<4; indx++) {
 aMyValues[indx] = Math.Sqrt( (Double) indx+1 );
}
```

Now, if you want to print out one of the values in your array, you can use `Console.WriteLine` like this:

```
Console.WriteLine(aMyValues[0].ToString());
Console.WriteLine(aMyValues[1].ToString());
Console.WriteLine(aMyValues[2].ToString());
Console.WriteLine(aMyValues[3].ToString());
```

What would happen if we attempted to assign a value to the `aMyValues[4]`? Remember that we created the array to holding four values, indexed from 0 to 3. So, index #4 is not defined…but let's just see what happens…

Arrays


```
using System;

class MainProgram {

 //Main program
 public static void Main() {

  //Create an array to hold 4 Doubles
  Double[] aMyValues = new Double[4];

  //Use a for-loop
  for(Int32 indx=0; indx<5; indx++) {
   aMyValues[indx] = Math.Sqrt( (Double) indx+1 );
  }

  //Print out values
  Console.WriteLine(aMyValues[0].ToString());
  Console.WriteLine(aMyValues[1].ToString());
  Console.WriteLine(aMyValues[2].ToString());
  Console.WriteLine(aMyValues[3].ToString());
  Console.WriteLine(aMyValues[4].ToString());

 }

}
```

Notice that the for-loop's stopping criteria is `indx<5` rather than `indx<4`. Here is the error message you will receive on the console:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the
array.    at MainProgram.Main()
```

We talk about error handling later in the presentation.

Arrays

Be aware of additional ways to initialize arrays. For example,

```
Int32[] aMyData = new Int32[] {1,2,3,4,5};
```

The example above initializes an array `aMyData` of `Int32`'s with the numbers 1 through 5. Thus, this is an array with five elements. Another way to do the same thing is:

```
Int32[] aMyData = {1,2,3,4,5};
```

For multi-dimensional arrays, we can do something similar:

```
Int32[,] aMyData = new Int32[,] { {1,2,3} , {4,5,6} };
```

For jagged arrays, discussed below, we can do something similar as well:

```
Int32[][] jaMyData = new Int32[3][];
jaMyData[0] = new Int32[] {1,2,3};
jaMyData[1] = new Int32[] {1,2,3,4,5};
jaMyData[2] = new Int32[] {1,2,3,4,5,6,7};
```

Arrays

Now, let's see how to use our `Point` structure in an array. Here is an example that computes the (x,y)-coordinate of each point on a polygon centered at the origin with each point a distance of 1 unit away from the origin.

```
using System;

struct Point {

 public Double X;
 public Double Y;

}

class MainProgram {

 //Main program
1 public static void Main(String[] sArg) {

2  Point[] aPolygon; //Create an array to hold the points of an S-sided polygon
  Int32 NumberOfSides; //Integer to hold number of sides
  Double Angle; //Double to hold angle in radians

  //Convert sArg[0] to an Int32..returns 0 if argument is null
  NumberOfSides = Convert.ToInt32(sArg[0]);

  //Check if the number of sides is 3 or greater.
  if ( NumberOfSides >= 3) {

   Console.WriteLine("Generating Polygon of {0:N} sides",NumberOfSides);

   //Initialize the aPolygon array to hold NumberOfSides elements.
3   aPolygon = new Point[NumberOfSides];
```

…continued on next slide…

Arrays


```
        //Generate the points of the polygon centered on a unit circle
        for(Int32 indx=0; indx<NumberOfSides; indx++) {

         //Determine the angle in radians...X-Axis at zero degrees is first
         Angle = (Math.PI/180) * ( (Double)(indx * (360/NumberOfSides)) );

         aPolygon[indx].X = Math.Cos(Angle);
         aPolygon[indx].Y = Math.Sin(Angle);

         Console.WriteLine("Side #{0:N0}: (X,Y)=({1:F3},{2:F3})",indx,aPolygon[indx].X,aPolygon[indx].Y);

        }

       }
       else {

        Console.WriteLine("Sorry...the value of " + sArg[0] + " is illegal. Enter in a value of 3 or greater.");

       }

      }

     }
```

4

5

6

1. Notice that we are now pulling in the command line arguments in a `String` array called `sArg`. In this case, there should be one entry on the line: `sArg[0]` containing the number of sides of the polygon.
2. Here we define the array `aPolygon` to hold `Point` structures, but we do not instantiate it until we know that the command line argument is valid.
3. Here is where we instantiate the array `aPolygon`.

Arrays

4. Here we compute the angle needed to generate the polygon. We convert from degrees to radians by multiplying by `Math.PI/180`.
5. These two lines compute the X- and Y-coordinate of the points on the polygon. Note that we are setting the public X and Y fields within the `aPolygon` structure for each specific point.
6. Here is where we write out the results for each iteration of the loop. Notice that we are using format strings within the text we want to output. Formatting strings are things like {0:N0}, {1:F2}, etc. We talk more about format strings later, but be aware that for each format within the quoted string, there is a corresponding variable. The number appearing to the left of the colon indicates which variable to print at that point within the format string.

```
C:\TEMP\test>test23 3
Generating Polygon of 3.0 sides
Side #0: (X,Y)=(1.000,0.000)
Side #1: (X,Y)=(-0.500,0.866)
Side #2: (X,Y)=(-0.500,-0.866)

C:\TEMP\test>test23 4
Generating Polygon of 4.0 sides
Side #0: (X,Y)=(1.000,0.000)
Side #1: (X,Y)=(0.000,1.000)
Side #2: (X,Y)=(-1.000,0.000)
Side #3: (X,Y)=(0.000,-1.000)

C:\TEMP\test>test23 5
Generating Polygon of 5.0 sides
Side #0: (X,Y)=(1.000,0.000)
Side #1: (X,Y)=(0.309,0.951)
Side #2: (X,Y)=(-0.809,0.588)
Side #3: (X,Y)=(-0.809,-0.588)
Side #4: (X,Y)=(0.309,-0.951)
```

Arrays

As you might suspect, you can create multi-dimensional arrays in C#. The syntax to create a two-dimensional array is as follows:

```
datatype[,] array-name = new datatype[#1,#2];
```

where datatype is the desired data type (such as `Double`, `Int32`, etc.)

array-name is the name of your array

$\#_1$ is the number of elements of the first dimension of your array

$\#_2$ is the number of elements of the second dimension of your array

Creating arrays of higher dimensions than two follow as you would suspect. Also, the initialization and usage follows just you would suspect.

Now, C# defines a property for arrays called `Rank` that tells you how many dimensions the array has. For example, if your array has three dimensions, the rank is 3. You can guess the rest:

```
Console.WriteLine(aMy3DArray.Rank); // Prints 3
```

Now, if you've every worked with multi-dimensional arrays, you know they're like pigs…memory speaking, of course. C# has defined another type of array called the *jagged array* which is just an array of arrays. Here's how you create a jagged array:

Arrays


```
datatype[][] jagged-array-name = new datatype[#1][#2];
```

where datatype is the desired data type (such as `Double`, `Int32`, etc.)

jagged-array-name is the name of your jagged array

$\#_1$ is the number of elements of the first dimension of your array

$\#_2$ is the number of elements of the second dimension of your array

Here is an example (see next slide):

Arrays


```
using System;
using System.Collections;

class MainProgram {

 //Main program
 public static void Main(String[] dArg) {

  Int32[][] jaMyData; //Create a jagged array to hold the points of an S-sided polygon

  //Initialize the first dimension to hold three slots (the first dimension) of ArrayLists.
  jaMyData = new Int32[3][];

  //Although we created the jagged array, we did not specify what each "jag" will hold
  jaMyData[0] = new Int32[] {1,2,3};
  jaMyData[1] = new Int32[] {1,2,3,4,5};
  jaMyData[2] = new Int32[] {1,2,3,4,5,6,7};

  //Print off the data
  for(Int32 I=0;I<3;I++) {

   Console.WriteLine("Jagged Array #" + I.ToString());

   for(Int32 J=0;J<jaMyData[I].Length;J++) {
    Console.WriteLine("jaMyData[{0:N0}][{1:N0}] = {2:F3}",I,J,jaMyData[I][J]);
   }

  }

 }

}
```

Enumerated Types

Recall in the Derived Classes section, we passed the parameter `pWhich` into our `ComputeArea` method. Within that code, `pWhich` was compared to the integers 1, 2 and 3, where 1 means "Rectangle Rule", 2 means "Trapezoidal Rule" and 3 means "Simpson's Rule".

Now, instead of using integers, we could have created an *enumerated type* (or *enumerator*) to associate meaningful descriptive text to the values 1, 2 and 3 and then used the descriptive text in the code instead of the values. Here is how we would create that enumerated type:

```
//Create an enum to hold the method names
enum eMethods {
 RectangleRule = 1,
 TrapezoidalRule,
 SimpsonsRule
}
```

Note that we use the `enum` keyword followed by the name of our enumerated type, `eMethods`. Within the braces, we specify a comma-delimited list of items. Now, `enum`s start, by default, at zero, so I placed an "=1" after the first item to indicate that the `enum` should start off at 1. From then on, each enumerated value is the next integer: 2, 3, 4, etc. If you don't necessarily want to assign *consecutive* integral values, you can force whatever value you want by using "=#" after the descriptive text.

Enumerated Types

You are not limited to integer values and you can specify which datatype you are using by placing a colon followed by the datatype after the name of the enumerator:

```
//Create an enum to hold the method names
enum eMethods : int {
 RectangleRule = 1,
 TrapezoidalRule,
 SimpsonsRule
}
```

You'll note that I used "`int`" instead of `Int32`. It seems that enumerators don't play well with the `System.`*`datatype`* names, so you'll have to use the low class type names (rough translation: use the primitive type names instead of the fancy Framework Class Library (FCL) type name): `(s)byte`, `(u)short`, `(u)int`, `(u)long`.

Note that you can specify either `public` or `internal` with an enumerator (to the left of the `enum` keyword): `internal enum eMethods : int {`. The default is `public`.

Now, if you are going to compare a variable declared as, say, `Int32` with an enumerated type, you will have to perform the conversion yourself. For example, here is our new `ComputeArea` method along with the `Main` program (see next slide…that parts of the `Main` program left unchanged have been removed from the example below):

Enumerated Types

```
public Double ComputeArea(Int32 pWhich) {

 if (pWhich == (Int32)eMethods.RectangleRule) {
  AREA_UNDER_CURVE =  RectangleRule();
  Console.WriteLine("Rectangle Rule");
 }
 else if (pWhich == (Int32)eMethods.TrapezoidalRule) {
  AREA_UNDER_CURVE =  TrapezoidalRule();
  Console.WriteLine("Trapezoidal Rule");
 }
 else if (pWhich == (Int32)eMethods.SimpsonsRule) {
  AREA_UNDER_CURVE =  SimpsonsRule();
  Console.WriteLine("Simpson's Rule");
 }
 else {
  AREA_UNDER_CURVE =  RectangleRule();
  Console.WriteLine("Rectangle Rule");
 }

 return(AREA_UNDER_CURVE);

}

}

class MainProgram {

 //Main program
 public static void Main() {

  …

  //Call the ComputeArea method to compute the area.
  areaRR = auc.ComputeArea((Int32)eMethods.RectangleRule);
  Console.WriteLine(areaRR.ToString());
  areaTR = auc.ComputeArea((Int32)eMethods.TrapezoidalRule);
  Console.WriteLine(areaTR.ToString());
  areaSR = auc.ComputeArea((Int32)eMethods.SimpsonsRule);
  Console.WriteLine(areaSR.ToString());

 }

}
```

We probably could have specified that our parameter pWhich is of the enumerated type and then we would not have to do the conversions to `Int32` in the MainProgram.

Enumerated Types

Now, one nice thing about enumerated types is that, given a value, you can get back the corresponding descriptive text instead of the value. This seems to only work if the values you've specified in the definition of your enumerator don't overlap at the bit level. Another way to put it is that each value you specify is of the form 2^n.

Let's talk about bits for…uh…a bit. Recall that you can represent decimal integers 1, 2, 3, 4… in binary as 001, 010, 011, 100, etc. Note, though, that the number 3 is represented as 011 which is the sum of 001 and 010. Clearly the number 3 cannot be represented in the form of 2^n. Here is an example enumerator for, say, credit cards:

```
//Create an enum to hold several credit card names
enum eCreditCards : int {
 None            =   0,
 Mastercard      =   1,
 Visa            =   2,
 Slate           =   4,
 Discover        =   8,
 AmericanExpress =  16,
 CapitalOne      =  32,
 ChaseFreedom    =  64,
 CitiForward     = 128,
 InkBold         = 256
}
```

Enumerated Types

Now, let's assume that you have a database table that contains customer IDs along with credit card usage. For shits-and-giggles, let's assume that our goal is to determine the *number of customers* who use only one credit card (and which those are), how many use only two credit cards (and which those are), and so on. You can think of this as a problem in determining intersections, like creating a Venn Diagram.

For example, let's assume that our database table looks like this:

CUSTOMER_ID CREDITCARD_ID SERVICE_DATE
PURCHASE_AMT PURCHASE_TYPE LOCATION_ID LOCATION_TYPE CUSTOMER_SCORE

I've made those columns up, so if you're in the credit card industry don't lose sleep over this table. The only columns that are of interest here are the CUSTOMER_ID and the CREDITCARD_ID…the rest can be ignored.

Now, for this example, we can assume that the CREDITCARD_ID column uses the same values in the enumerator. If not, you can easily transform it to match our enumerator (or whatever your enumerator is) by using a CASE Statement in your SQL.

Enumerated Types

Here is some (very fake) data that does not use our enumerator values:

CUSTOMER_ID	CREDITCARD_NAME
1	Mastercard
2	Mastercard
2	Visa
3	Discover
4	Mastercard
4	Visa

As you see, we have reduced down the data to the DISTINCT CUSTOMER_ID by CREDITCARD_NAME level. Now, in the data above, we have two customers who only use one single credit card (CUSTOMER_IDs 1 and 3); and, we have two customers who use two cards (CUSTOMER_IDs 2 and 4)

Now, how would you determine this information normally? You would probably create several small tables, each one holding a single credit card's data. Then, you would perform a series of INNER JOINs on each table (two at a time) in order to determine which customers used two credit cards. Then again, three at a time, and so one. That's A LOT of SQL code!!

Enumerated Types

Here is another way to do this by using the fact that our enumerator values are formed as 2^n.

For example, in our enumerator we've set Mastercard to the value 1 (2^0), Visa to the value 2 (2^1), and Slate to the value 4 (2^2). Now, if a customer uses both Mastercard and Visa, we add the values 1 and 2 to get 3. In binary form, 3 is 011 which is made up of 001 (the number 1 indicating Mastercard) and 010 (the number 2 indicating Visa). If the customer uses all three of these cards, we have 1+2+4 = 7 which is 111 in binary (100 indicating Slate, 010 indicating Visa, 001 indicating Mastercard). Here is the SQL to do that (for the first few credit cards anyway):

```
SELECT CC_BITFLAG,COUNT(DISTINCT CUSTOMER_ID) AS CUSTOMER_CNT
 FROM (
       SELECT CUSTOMER_ID,SUM(CREDITCARD_ID) AS CC_BITFLAG
        FROM (
              SELECT CUSTOMER_ID,
                     CASE
                      WHEN CREDITCARD_NAME='MASTERCARD' THEN 1
                      WHEN CREDITCARD_NAME='VISA' THEN 2
                      WHEN CREDITCARD_NAME='SLATE' THEN 4
                     END CASE AS CREDITCARD_ID
               FROM (
                     SELECT DISTINCT CUSTOMER_ID, CREDITCARD_NAME
                      FROM CREDIT_INFO_TABLE
                    )
             )
        GROUP BY CUSTOMER_ID
      )
 GROUP BY CC_BITFLAG;
```

Enumerated Types

Now the SQL in RED just creates a distinct list of CUSTOMER_IDs and CREDITCARD_NAMES. This is a must…since the uniqueness at this level will prevent the SUM aggregate function from screwing things up. Next, the code in PURPLE replaces the text with the enumerator values. The code in GREEN sums up the CREDITCARD_ID to the CUSTOMER_ID level. YES! You are not misreading the SQL…check the explanation on the previous slide! Finally, the code in BLACK just computes the number of distinct CUSTOMER_IDs for each combination of CC_BITFLAG. Here is what the PURPLE code returns:

CUSTOMER_ID	CREDITCARD_NAME	CREDITCARD_ID
1	Mastercard	1
2	Mastercard	1
2	Visa	2
3	Discover	8
4	Mastercard	1
4	Visa	2

Enumerated Types

Here is what the GREEN code returns:

CUSTOMER_ID	CC_BITFLAG
1	1
2	3
3	8
4	3

Finally, here is what the BLACK code returns:

CC_BITFLAG	CUSTOMER_CNT
1	1
3	2
8	1

What this tells you is that only ONE customer used CREDITCARD_ID 1 (Mastercard ONLY), two customers use CREDITCARD_ID 3 (Mastercard and Visa) and only ONE customer uses CREDITCARD_ID 8 (Discover). Naturally, you'll wind up with a lot more rows than this if you are using REAL data!!

Enumerated Types

Hmmm…I may have strayed a bit from C#, but here's where I bring it all back together. Let's assume that the results have been pulled back from the database (using ADO.NET) and reside in an array. We can use the data in the array to create the text descriptions:


```
using System;

//Create an enum to hold several credit card names
enum eCreditCards : int {
 None            =   0,
 Mastercard      =   1,
 Visa            =   2,
 Slate           =   4,
 Discover        =   8,
 AmericanExpress =  16,
 CapitalOne      =  32,
 ChaseFreedom    =  64,
 CitiForward     = 128,
 InkBold         = 256
}

class MainProgram {

 //Main program
 public static void Main() {

  //Create an array to hold the BLACK code's final data.
  //You'd pull this back from the database using ADO.NET.
  Int32[] aCC_BITFLAG = new Int32[] {1,3,8};

  //Create a variable of type eCreditCards
  eCreditCards eCC = eCreditCards.None;

  //Write out the descriptive text instead of the number 1,3, and 8.
  for(Int32 indx=0;indx<3;indx++) {

   //Convert the Int32 value to the enum type eCreditCards
   eCC = (eCreditCards) aCC_BITFLAG[indx];

   //Use the "F" flag to translate to textual values.
   Console.WriteLine(eCC.ToString("F"));

  }

 }

}
```

Enumerated Types

1. Here we create out enumerated type, `eCreditCards`
2. After reading in the data into an array, we create a variable of type `eCreditCards` (an `enum` type). This is initialized to `eCreditCards.None`.
3. Pull the current array value using the loop index indx and then convert it to type `eCreditCards` by casting. This ensures that C# knows how to handle it in the next step.
4. Finally, write out the value using the "F" flag of the `ToString` method for the variable `eCC`. Since `eCC` is an `enum` type, the numeric value is converted to the descriptive text specified in the enumerator `eCreditCard`.

Here are the results from running the program:

```
C:\TEMP\test>test26
Mastercard
Mastercard, Visa
Discover
```

Handling Exceptions

Normally, all of our code runs perfectly, but occasionally a stray gamma ray from the sun (or from a nearby starship's matter-antimatter engine) causes an unpredictable glitch to occur within our perfect program. In order to prevent our program from crashing, we can use *exception handling* to capture errors and take appropriate steps to work-around the error, or exit gracefully with a pleasing and informative error message.

In C#, exception handling is handled via the try-catch-finally block which looks something like this:

```
try {
 //perform some code that could error-out
}
catch (catch-type1) {
 //handle exceptions of catch-type1
}
catch (catch-type2) {
 //handle exceptions of catch-type2
}
catch {
 //handle all other errors not handled above
 throw; //usually this generic catch-block rethrows the error
}
finally {
 //code in this block is ALWAYS executed regardless of an error OR NOT!!
 //Maybe close database connection or suspend communications with Borg ship.
}
//Code here is run if no error occurred, OR an error occurred and not rethrown!
```

Handling Exceptions

As you see, you start off with the `try` keyword and you place the code you think might give an error, such as a failed connection to a database, etc. Then you follow the try-block with one or more catch-blocks. Each catch-block can be coded to catch a specific error (catch-type#), or can be a catch-all (tee-hee!) catch-block with no catch-type#. You can place a finally-block at the end to run code regardless of whether an error occurred ***or not***. The finally-block can close an open file, close a connection to a database, etc. You can omit the finally-block if you don't think you need it.

Note that, if you have a generic catch-block, you MUST place it AFTER all of the catch-blocks that have defined catch types. That is, more specific catch-blocks come first followed by the generic catch-block last.

Now, the catch types are either `System.Exception` or are derived from `System.Exception`.

Now, you may be wondering what would happen if you placed a method within a try-block and an error occurred *within that method*. Well, if you have defined a try-catch-finally block within your method AND you capture the error there, then all is well…you caught the error. But, if you did NOT catch the error, the Common Language Runtime (CLR) would *continue to search up the call stack* for catch-blocks that match the exception. If an appropriate catch-block is found, then all is well…if not, the CLR throws an exception and an error appears either at the command line or a dialog box pops up giving your user the bad news!

Handling Exceptions

Recall in the section on Arrays, we had an example of what would happen if we tried to add a fifth value to an array defined to hold only four items. An error occurred, specifically, this error occurred:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the
array.   at MainProgram.Main()
```

As you see, the CLR is telling you that the error `System.IndexOutOfRangeException` has occurred. Let's redo this code by placing a try-catch-finally block in our code. See next slide for complete code. You'll note that within the catch-block we decrement our index counter variable iIndexCounter in order to avoid another error message within the finally-block when we print the values to the screen. Note that we would have placed the `Console.WriteLine()` code below the finally-block and ditched the finally-block completely…your call.

Handling Exceptions


```
using System;

class MainProgram {

 //Main program
 public static void Main() {

  //Create an array to hold 4 Doubles
  Double[] aMyValues = new Double[4];

  //Create integer to hold array counts
  Int32 iIndexCount = 5;

  //Use a for-loop
  try {

   for(Int32 indx=0; indx<iIndexCount; indx++) {
    aMyValues[indx] = Math.Sqrt( (Double) indx+1 );
   }

  }
  catch (System.IndexOutOfRangeException) {

   Console.WriteLine("A System.IndexOutOfRangeException occurred within your code, you dolt!");
   iIndexCount--; //Ooops...we specified one too many...reduce by one or the code that follows will error out!!

  }
  finally {

   //Print out values...could have gone below and ditched the finally-block.
   for(Int32 indx=0; indx<iIndexCount; indx++) {
    Console.WriteLine(aMyValues[indx].ToString());
   }

  }

 }

}
```

Handling Exceptions

Now, there are many exceptions available for you to use. See the `SystemException`, `ApplicationException` and `Exception` classes on Microsoft's website.

Now, some authors add a variable to the exception catch type instead of just putting the catch type itself in parentheses. For example,

```
catch (System.IndexOutOfRangeException e) {

 Console.WriteLine(e.Message);
 iIndexCount--; //Ooops...we specified one too many...

}
```

As you can see, we set the variable `e` to be of an exception type and then used its `Message` property to write out the error message. Another nice property is the `StackTrace` property which shows the methods that were called that led up to the error:

```
catch (System.IndexOutOfRangeException e) {

 Console.WriteLine(e.Message);
 Console.WriteLine(e.StackTrace);
 iIndexCount--; //Ooops...we specified one too many...

}
```

Handling Exceptions

Another nice exception handling feature is that you can throw your own error messages instead of just handling Microsoft's defined exceptions. You might do this if you are attempting to ping a database and you discover that the database is not responding. At that point, you can throw your own exception:

```
throw new Exception("The freakin' database isn't responding to pings, dammit!");
```

You can then catch this exception in a generic catch-block and use `e.Message` to get your descriptive error message.

Note that we mentioned that you can re-thrown an exception using the throw keyword. This has the effect of throwing the *same* exception up the call stack *which caused the original exception*.

You probably won't be surprised – given that an exception is derived from the `System.Exception` class – that you can create your own exception class by deriving from the `System.Exception` class. We don't into this here, but be aware that it exists.

The `is` and `as` Operators

I've seen in several places where programmers check the type of their objects against a specific class – say, with an `if` statement – before proceeding to the code within the `if` statement. I assume that this is just prudent coding. C# has two operators to perform this type of check: `is` and `as`.

The `is` Operator is used like this:

```
MyClass mc = new MyClass();
…code goes here…
if (mc is MyClass) {
 //perform code using mc
}
…code goes here…
```

The `is` Operator above checks whether `mc` is compatible with type `MyClass`. Note that the `is` Operator returns `true` or `false` and never throws an exception.

The `as` Operator allows you to perform a type check *as well as* the creation of a new object at the same time. If this fails, a `null` is returned; otherwise you get a pointer to the object:

```
MyObject mo = new Object();
MyClass mc = mo as MyClass;
if (mc != null) {
 …
}
```

Apparently, using `as` and comparing against `null` is very fast compared to performing an `is` operation followed by object instantiation.

C# Compiler Command Line Arguments

In this section, we explain some of the command line arguments you can use when compiling C# programs at the command line. Here is what csc/? shows:

```
                        Visual C# 2010 Compiler Options

                        - OUTPUT FILES -
/out:<file>                     Specify output file name (default: base name of
                                file with main class or first file)
/target:exe                     Build a console executable (default) (Short form:
                                /t:exe)
/target:winexe                  Build a Windows executable (Short form:
                                /t:winexe)
/target:library                 Build a library (Short form: /t:library)
/target:module                  Build a module that can be added to another
                                assembly (Short form: /t:module)
/delaysign[+|-]                 Delay-sign the assembly using only the public
                                portion of the strong name key
/doc:<file>                     XML Documentation file to generate
/keyfile:<file>                 Specify a strong name key file
/keycontainer:<string>          Specify a strong name key container
/platform:<string>              Limit which platforms this code can run on: x86,
                                Itanium, x64, or anycpu. The default is anycpu.

                            - INPUT FILES -
/recurse:<wildcard>             Include all files in the current directory and
                                subdirectories according to the wildcard
                                specifications
/reference:<alias>=<file>       Reference metadata from the specified assembly
                                file using the given alias (Short form: /r)
/reference:<file list>          Reference metadata from the specified assembly
                                files (Short form: /r)
/addmodule:<file list>          Link the specified modules into this assembly
/link:<file list>               Embed metadata from the specified interop
                                assembly files (Short form: /l)
```

…continued on next slide…

C# Compiler Command Line Arguments


```
                        - RESOURCES -
/win32res:<file>              Specify a Win32 resource file (.res)
/win32icon:<file>             Use this icon for the output
/win32manifest:<file>         Specify a Win32 manifest file (.xml)
/nowin32manifest              Do not include the default Win32 manifest
/resource:<resinfo>           Embed the specified resource (Short form: /res)
/linkresource:<resinfo>       Link the specified resource to this assembly
                              (Short form: /linkres)
                              Where the resinfo format is <file>[,<string
                              name>[,public|private]]

                        - CODE GENERATION -
/debug[+|-]                   Emit debugging information
/debug:{full|pdbonly}         Specify debugging type ('full' is default, and
                              enables attaching a debugger to a running
                              program)
/optimize[+|-]                Enable optimizations (Short form: /o)

                        - ERRORS AND WARNINGS -
/warnaserror[+|-]             Report all warnings as errors
/warnaserror[+|-]:<warn list> Report specific warnings as errors
/warn:<n>                     Set warning level (0-4) (Short form: /w)
/nowarn:<warn list>           Disable specific warning messages

                        - LANGUAGE -
/checked[+|-]                 Generate overflow checks
/unsafe[+|-]                  Allow 'unsafe' code
/define:<symbol list>         Define conditional compilation symbol(s) (Short
                              form: /d)
/langversion:<string>         Specify language version mode: ISO-1, ISO-2, 3,
                              or Default

                        - MISCELLANEOUS -
@<file>                       Read response file for more options
/help                         Display this usage message (Short form: /?)
/nologo                       Suppress compiler copyright message
/noconfig                     Do not auto include CSC.RSP file
```

…continued on next slide…

C# Compiler Command Line Arguments


```
                        - ADVANCED -
/baseaddress:<address>          Base address for the library to be built
/bugreport:<file>               Create a 'Bug Report' file
/codepage:<n>                   Specify the codepage to use when opening source
                                files
/utf8output                     Output compiler messages in UTF-8 encoding
/main:<type>                    Specify the type that contains the entry point
                                (ignore all other possible entry points) (Short
                                form: /m)
/fullpaths                      Compiler generates fully qualified paths
/filealign:<n>                  Specify the alignment used for output file
                                sections
/pdb:<file>                     Specify debug information file name (default:
                                output file name with .pdb extension)
/nostdlib[+|-]                  Do not reference standard library (mscorlib.dll)
/lib:<file list>                Specify additional directories to search in for
                                references
/errorreport:<string>           Specify how to handle internal compiler errors:
                                prompt, send, queue, or none. The default is
                                queue.
/appconfig:<file>               Specify an application configuration file
                                containing assembly binding settings
/moduleassemblyname:<string>    Name of the assembly which this module will be a
                                part of.
```

Now, you can use these options to not only create an executable file, but to create DLLs. Let's create a DLL that contains a class that has a static method to add one to a given number. That program will be compiled using the `/target:name` command line option. Then, a program containing a call to that method will be created and compiled using the `/references:name` command line option. Then we will execute that program to see if it works!

C# Compiler Command Line Arguments

Here is the code for what will be our DLL:

```
using System;

public class MyNumberClass {

 static public Int32 AddOne(Int32 pNum) {
  return(pNum+1);
 }

}
```

Here is how you compile it to make the DLL `test43.dll`:

```
csc /target:library test43.cs
```

Here is the code for what will be our test executable:

```
using System;

class MainProgram {

 //Main program
 public static void Main() {

  Console.WriteLine(MyNumberClass.AddOne(5));

 }

}
```

Here is how you compile it to make the executable using the DLL:

```
csc /references:test43.dll /target:exe test43MP.cs
```

C# Compiler Command Line Arguments

Note that we do not need the `/target:exe` since that is the default. When we run this, we get back the number 6 at the command line.

Be aware that you can name your executable or dynamic link library something other than the default name by using the `/out:`*`name`* command line option. You can enable optimizations by using `/optimize+` or just `/o`. You can also compile to a specific platform by using the `/platform:`*`name`*, where name can be one of the following: `x86`, `Itanium`, `x64` or `anycpu`. The default is `anycpu`.

Another option that I find useful is the `/debug+` switch which forces error messages to tell you what line in your code caused the error. Very useful!

Finally, in the next presentation we talk about attributes and pre-processor directives. In order to define pre-processor variables you use the `/define:`*`name`* command line option.

What Next?

This completes the *C# Programming II* presentation. The next step is to move on to the advanced lecture (*C# Programming III*), or to peruse the individual namespace presentations (*C# Programming IV-#*).

In *C# Programming III*, we delve into some advanced topics such as interfaces, delegates, collections, etc.

In *C# Programming IV-#*, we look at specific classes within specific namespaces such as the `System` namespace, the `System.Data` namespace, etc.

Don't forget to peruse my *Self-Inflicted C# Projects* in the Documents section of the Sheepsqueezers website (www.sheepsqueezers.com).

References

Click the book titles below to read more about these books on Amazon.com's website.

- Introducing Microsoft LINQ, Paolo Pialorsi and Marco Russo, Microsoft Press, ISBN:9780735623910
- LINQ Pocket Reference, Joseph Albahari and Ben Albahari, O'Reilly Press, ISBN:9780596519247
- Inside C#, Tom Archer and Andrew Whitechapel, Microsoft Press, ISBN:0735616485
- C# 4.0 In a Nutshell, O'Reilly Press, Joseph Albahari and Ben Albahari, ISBN:9780596800956
- The Object Primer, Scott W. Ambler, Cambridge Press, ISBN:0521540186
- CLR via C#, Jeffrey Richter, Microsoft Press, ISBN:9780735621633

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.