

Lynda C# Essentials 2012

<http://msdn.microsoft.com/library>

Installation

VS2010

- Install NuGET
- Selenium .NET libraries
- NUnit .NET libraries

Keyboard Shortcuts

- Build = C-S-B \ F6
- Run = F5

Operator Precedence

1 Multiplicative:	* / %
2 Additive:	+ -
3 Comparison and Type Testing:	< > <= >= is as
4 Equality:	== !=
5 Conditional AND:	&&
6 Conditional OR:	
7 Ternary:	?:
8 Assignment:	= *= /= %= += -=

Chapter 01

Project v.s. Solution

- Solution is TOP level project that can contain other projects

Creating a New Project

File, New, Visual C#, Console Application



Project Components

- AssemblyInfo.cs - Project Metadata for .NET framework
- References - .NET libraries, will change as project grows
-

VS2010 Tour

View\ErrorList:



Tools\Options...



New Project Code:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Hello World");

            Console.ReadLine();

        }

    }

}
```

```
}  
  
}  
  
}
```

Chapter 02



Overview of C#

- compiled code
- OO programming language: almost everything is an object
- Everything is in classes, no global variables
- Foundation language for many MS platform technologies
- Uses CLR (Common Language Runtime)

Declaring Variables

- type name
- helps compiler optimize code
- `int myCounter;`
- `string message;`
- `int myCounter = 200;`
- `string message = "Hello World";`

Namespaces

Namespace is just a collection of related classes (or other namespaces)



Console Applications

Which .NET pieces to use in the program: using statement

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;
```

Namespace to wrap program in its own area to avoid collisions with other projects and classes

```
namespace StructureCSharp

{
```

Classes creates an object, the fundamental building block of CSharp

```
class Program

{
```

Main function is required in each class: it will accept multiple arguments

```
static void Main(string[] args)

{
```

// These are statement blocks

```

    }

}
}

```

Namespaces, Using Statements and 'Fully Qualified Paths' to objects

Namespaces provide a relative structure to project and its classes

'Using' Statement includes the namespace in such way that access objects can be shorter

MUST use base classes (i.e. System...as first item in the object's path if no 'using' statement

Ex: These are the same:

```
using System;
```

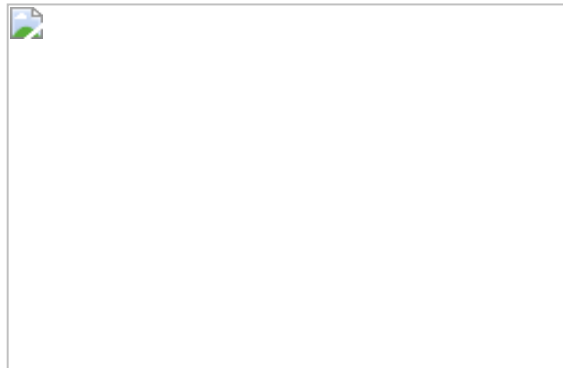
```
Console.WriteLine("Hello World");
```



AND

```
System.Console.WriteLine("Hello World");
```

Compiling/Build

Compiled projects saved to 'bin' folder in the project



- *.sln = Solution Metadata
- *.pdb = .NET database for managing the solution
- Warning: 
- Error: 

Errors v.s. Output

Output has more info:

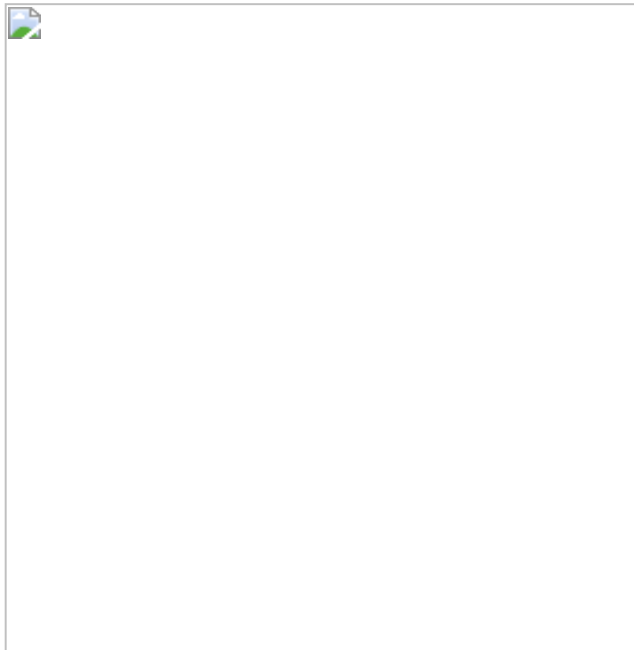
```
----- Build started: Project: BuildAndRun, Configuration: Debug x86 -----
```

```
D:\DEVL\C#\Solutions\BuildAndRun\BuildAndRun\Program.cs(12,17): warning CS0219:  
The variable 'myInt' is assigned but its value is never used
```

```
Compile complete -- 0 errors, 1 warnings
```

```
BuildAndRun ->  
D:\DEVL\C#\Solutions\BuildAndRun\BuildAndRun\bin\Debug\BuildAndRun.exe
```

```
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =====
```



- Assembly = this is physical peice of code somewhere...
- Default namespace
- Target Framework = .NET target
- Output Type = what kind of application building



Build Options



Chapter 03


```
Console.WriteLine("Hello World!");
```

```
Console.ReadLine();
```

Building Strings in C# (Concatenizing)

- + works, but not preferred

```
String.Format()
```

```
String.Format("There are {0} seconds in a year", i);
```

```
String.Format("There are {0} seconds, [1] minutes in a year", i, j);
```

- 'Console.WriteLine' automatically uses the String.Format() internal
- Pass it a 'format string'

```
Console.WriteLine("There are {0} seconds in a year", mySeconds);
```

Commenting

```
/*
```

```
*/
```

Conditional Statements

IF THEN ELSE

```
if ( ) { }
```

```
else if ( ) { }
```

```
else { }
```

OR

```
if ( ( ) || ( ) ) { }
```

AND:

```
if ( ( ) && ( ) ) { }
```

Switch Statement

- 'break' is required for EACH case statement

```
switch ( value ) {  
  
    case: value1:  
  
        // run some code  
  
        break;  
  
    case valueN:  
  
        //run some code  
  
        break;  
  
    default:  
  
        //run some code  
  
        break;  
  
}
```

Operators and Expressions

Operators = *, +

Operand = operated on

Arithmetic Operators: + _ / * %

Assignment Operators: =

Comparison Operators: == != < > >= <=

Logical Operators: && || !

```
if ( a==b && c == d) {}
```

```
if ( a==b || c == d) {}
```

```
if (!someBooleanValue){}
```

Increment / Decrement Operators:

```
a = a + 1;
```

```
a += 1;
```

```
a++; (postfix)
```

```
++a; (prefix)
```

```
a = a - 1;
```

```
a -= 1;
```

```
a--;
```

```
--a;
```

Prefix(++a) and Postfix(a++)

```
int a = 10;
```

```
Console.WriteLine("The Value of a is: {0}", ++a); Output is 11
```

```
Console.WriteLine("The Value of a is: {0}", a++); Output is 10
```

Type Testing Operators: is as

is operator returns true if a given object IS a certain type

as operator converts a given object to another type (if possible)

```
void myFunctions(object obj) {  
  
    // if type is Employee  
  
    if (obj is Employee)  
  
    {  
  
        // not clear on the purpose????  
  
        Employee emp = obj as Employee; // null if can't be done
```

```

        string name = emp.FirstName;

        ...

    }

```

Ternary Operator: ?:

Very compact if/else statement
(condition) ? true : false

```

int price1;

int price2;

int lowPrice;

// lowPrice =  if (condition) ? true : false

lowPrice = (price1<price2) ? price1 : price2;

if (price1 < price2)

    lowestPrice = price1;

else

    lowestPrice = price2;

```

Operator Precedence

9 Multiplicative:	* / %
10 Additive:	+ -
11 Comparison and Type Testing:	< > <= >= is as
12 Equality:	== !=
13 Conditional AND:	&&
14 Conditional OR:	
15 Ternary:	?:
16 Assignment:	= *= /= %= += -=

Using Constants and Enumerations: CONST/ENUM

Constants

```
if (someVar == 32) {} ← what is 32?
```

```
const int FREEZING = 32;
```

```
if (someVar == FREEZING){}
```

Enumerations

Default type is int (Integer)

```
const int FREEZING = 32;
```

```
const int LUKEWARM = 65;
```

```
const int HOT = 105;
```

```
const int BOILING = 212;
```

```
enum Temperatures {  
  
    REALLYCOLD,          // defaults to 0  
  
    FREEZING = 32,  
  
    FREEZING2,          //defaults to 33  
  
    LUKEWARM = 65,  
  
    HOT = 105,  
  
    BOILING = 212  
  
}
```

Re-Typing Enumerations

```
enum Temperatures : byte {} //max value of 255
```

```
enum Temperatures : string {} //string
```

```
enum Temperatures
```

```
{
```

```
    FREEZING = 32,
```

```
    LUKEWARM = 65,
```

```
    ROOMTEMP = 72,
```

```
    HOT = 105,
```

```
    BOILING = 212
```

```
}
```

```
static void Main(string[] args)
```

```
{
```

```
    int myTemp = 30;
```

```
    if (myTemp >= (int)Temperatures.FREEZING && myTemp <=
(int)Temperatures.BOILING)
```

```
    {
```

```
        Console.WriteLine("At this temperature, water is a liquid");
```

```
    }
```

```
    else
```

```
    {
```

```
        Console.WriteLine("Water is not a liquid at this temperature");
```

```
    }
```

```
        Console.ReadLine();  
  
    }  
  
}
```

Loops

While

```
while ( true ) { }
```

MAY NOT Execute

```
while ( true ) {  
  
}
```

Do While

```
do { } while (true);
```

Executes AT LEAST ONCE

```
do  
  
{  
  
} while ();
```

For

```
for ( count ) { }
```

Foreach

```
foreach ( element in a set ) { }
```

Continue

```
if ( true ) {  
  
    continue; // skip rest of the code in the loop and re-iterate  
  
}
```

Break

```
if (true) {  
  
    break; // breaks out of the loop  
  
}
```

Example

```
int myVal = 15;  
  
// basic while loop  
  
Console.WriteLine("Basic while() loop:");  
  
while (myVal < 20) {  
  
    Console.WriteLine("myVal is currently {0}", myVal);  
  
    // careful! Always make sure that the loop has some way to exit  
  
    // or you'll end up with an Infinite Loop!  
  
    myVal += 3;  
  
}  
  
Console.WriteLine();  
  
  
// the do-while loop  
  
Console.WriteLine("The do-while() loop:");  
  
do  
  
{  
  
    Console.WriteLine("myVal is currently {0}", myVal);  
  
    myVal += 3;
```



```
} while (myVal < 20);

Console.WriteLine();

// the for loop

Console.WriteLine("The for loop:");

for (int i = 0; i < myVal; i += 5)

{

    Console.WriteLine("i is currently {0}", i);

}

Console.WriteLine();

// using the continue and break keywords

Console.WriteLine("Using break and continue :");

for (int i = 0; i < 10; i++)

{

    if (i == 5)

        continue; // skip the rest of the loop

    if (i == 9)

        break;

    Console.WriteLine("i is currently {0}", i);

}
```

Functions and Methods

- OO languages Function = Method
- Always part of an Object
- No global functions
- No specific order of function declarations needed

Function Structure

```
passBackDataType functionName ( parameter list) {}
```

Returns int value

```
int myFunction (int x, data d ) { }
```

Returns NO value

```
void myFunction (int x, data d ) { }
```

Function Declaration

```
int myFunction (int x, data d ) {  
  
    //code  
  
    Console.WriteLine("We're in a function!");  
  
    return someValue; //return int value  
  
}
```

Example

```
int result1;  
  
result1 = formula(14);  
  
Console.WriteLine("The result is: {0}", result1);
```

```
static int formula(int theVal)

{

    return (theVal * 2) / 3 + 15;

}
```

Chapter 04

Data Types

Can't change data type of a variable
Must use type conversion methods myInt.To

Primitive Data Types



Nullable Types

Nullable types can represent the full range of values of their normal underlying type, along with the additional value of 'null'

```
bool b;           //normal boolean declaration
```

```
bool? b;          //boolean OR null
```

```
int? i;           //integer or null
```

Declaring Variables

```
int myVariable = 100;
```

```
string myLabel = "Some String";

object someObj;

char myChar = '8';

string myString = "  This is a message with a \n newline and spaces in it.  ";

int myInt = 100;

string myString2 = myInt.ToString();

// Do some character tests

Console.WriteLine("Calling char.IsUpper: {0}", char.IsUpper(myChar));

Console.WriteLine("Calling char.IsDigit: {0}", char.IsDigit(myChar));

Console.WriteLine("Calling char.IsLetter: {0}", char.IsLetter(myChar));

Console.WriteLine("Calling char.IsPunctuation: {0}", char.IsPunctuation(myChar));

Console.WriteLine("Calling char.IsWhiteSpace: {0}", char.IsWhiteSpace(myChar));

Console.WriteLine("Calling char.ToUpper: {0}", char.ToUpper(myChar));

Console.WriteLine("Calling char.ToLower: {0}", char.ToLower(myChar));

Console.WriteLine();

// do some string tests

Console.WriteLine("Calling string.Trim: {0}", myString.Trim());

Console.WriteLine("Calling string.ToUpper: {0}", myString.ToUpper());

Console.WriteLine("Calling string.ToLower: {0}", myString.ToLower());
```

```
Console.WriteLine("Calling string.IndexOf: {0}", myString.IndexOf("a"));
```

```
Console.WriteLine("Calling string.LastIndexOf: {0}",  
myString.LastIndexOf("and"));
```

```
Console.ReadLine();
```

System.Object: (ALMOST) Everything is an Object

- `System.Object.Int32 = int`
- `System.Object.String = string`
- Since everything is an object, can do type testing and perform operations based on result
- can convert **int** object and pass to function as a **string**.

```
int i = 0;
```

```
int i = new int();
```

```
i.GetType(); // returns System.Int32;
```

```
i.ToString();
```

```
i.Equals(j);
```

is operator

```
int myFunction(object obj) {  
  
    if (obj is int) {  
  
        int i = (int)obj;  
  
        // operate on the integer  
  
    }  
  
    else if (obj is string) {
```

```

        string s = (string)obj;

        // operate on the string

    }

}

```

Working with Integer Numbers

byte	8	0 to 255
sbyte	8	-128 to 127
short	16	-32,768 to 32,767
ushort	16	0 to 65,535
int	32	-2.1B to 2.1B
uint	23	0 to 4.3B
long	64	-9BIG to 9BIG
ulong	64	16BIGBIG

Floating Point Numbers (Base10 or Base2)

float	0.0f	7 digits of precision
double	0.0d	15-16 digits of precision (base 2 numbers)
decimal	0.0m	28-29 digits of precision (base 10 numbers)

Special Floating Point Values

NaN	Not a number
PositiveInfinity	0 < Infinity
NegativeInfinity	0 > Infinity

Double v.s. Decimal

double	0.0d	15-16 digits of precision (base 2 numbers)
decimal	0.0m	28-29 digits of precision (base 10 numbers)



- double Native to CPU and FAST
- decimal up to 10x slower than double
- double for Scientific Calculations
- decimal for Financial Calculations

Working with Characters and Strings

```
char myChar = 'a'; //MUST USE SINGLE QUOTE
```

```
string myString = "a string"; //MUST USE DOUBLE QUOTE
```

`\n` newline escape sequence

```
string myString = "here's a \n linebreak";
```

@ \ are not escape chars, treat as \

```
string myDirectory = "C:\\mydir\\somefile.txt";
```

```
string myDirectory = @"C:\mydir\somefile.txt";
```

```
char c='a';
```

```
char.IsUpper(c);
```

```
char.IsDigit(c);
```

```
string s="my message.";
```

```
s.ToUpper();
```

```
s.Trim();
```

- String are immutable - can't be changed once created
- Performance penalty as a result

```
string result = "";
```

```
for (int c=0; c < 1000; c++) {
```

```
    result += "some other string"; // creates and destroys new string
```

```
}
```

- Use built in 'String Builder' class instead

```
StringBuilder sb = new StringBuilder();

for (int c=0; c < 1000; c++) {

    sb.Append("some other string"); //StringBuilder

}

result = sb.ToString();
```

Char/String Methods

```
// Do some character tests

Console.WriteLine("Calling char.IsUpper: {0}", char.IsUpper(myChar));

Console.WriteLine("Calling char.IsDigit: {0}", char.IsDigit(myChar));

Console.WriteLine("Calling char.IsLetter: {0}", char.IsLetter(myChar));

Console.WriteLine("Calling char.IsPunctuation: {0}", char.IsPunctuation(myChar));

Console.WriteLine("Calling char.IsWhiteSpace: {0}", char.IsWhiteSpace(myChar));

Console.WriteLine("Calling char.ToUpper: {0}", char.ToUpper(myChar));

Console.WriteLine("Calling char.ToLower: {0}", char.ToLower(myChar));

Console.WriteLine();

// do some string tests

Console.WriteLine("Calling string.Trim: {0}", myString.Trim());

Console.WriteLine("Calling string.ToUpper: {0}", myString.ToUpper());

Console.WriteLine("Calling string.ToLower: {0}", myString.ToLower());
```



```
Console.WriteLine("Calling string.IndexOf: {0}", myString.IndexOf("a"));
```

```
Console.WriteLine("Calling string.LastIndexOf: {0}",  
myString.LastIndexOf("and"));
```

Variable Scope

```
int myFunction ()N {  
  
    int x = 10;  
  
    for (i=0; i<10; i++) {  
  
        int y = x + 20; //ok  
  
        // other code  
  
    }  
  
    x++; //ok  
  
    y += 20; // error - y is not available here  
  
}
```

- JavaScript has function level scoping, lifts all variables to top (same scope)
- C#, C, ObjectiveC - Scoping counts
- No duplicate var names in nested functions

```
int var1 = 10;  
  
for () {  
  
    int var1 = 20; // error!!!  
  
}
```

Type Conversion

Converting one type of variable to another is called casting

```
int i = 10;
```

```
float f = 20.0f;
```

```
f = i; // OK - implicit cast from int to float ( no data loss)
```

```
i = f; // ERROR - explicit cast from float to int needed
```

```
i = (int)f; // OK now - explicit type cast
```

Chapter 05

Object Orientation and C#

- Class
- Interface
- Object

Class

- blueprint or a description of what an object looks like
- what kinds of data an object of this type hold
- what kinds of things that an object of this type can do
- Class is NOT the object itself
- Define own classes AND use .NET provided one
-



- Classes define two major things (called members)
 - Fields and Properties - kinds of data the object holds and works with
 - Methods - what the object can do (functionality)
- Classes can inherit attributes and methods from other classes
 - Subclass inherits from Superclass



- Classes are used to create individual objects - called instances



- Abstracts ideas away from a specific implementation
 - Allow focus on the core parts for the program
- Encapsulates program functionality into understandable pieces
 - Related data and logic are all kept in one place
- Helps organize a program into natural parts that can be extended
 - Sometimes referred to as factoring
 - Mirrors real-world constructs and ideas

Defining a Class

```
class myClass {  
  
    //fields to contain the class data  
  
    int myInteger;  
  
    string myMessage;  
  
  
    // constructor SAME NAME AS CLASS  
  
    // .NET will create one for you  
  
    // Runs automatically upon instantiation  
  
    public myClass (string defaultMsg) {
```

```

        myMessage = defaultMessage;

    }

    //methods that define functionality

    public int myFunction() {

        return myInteger;

        // more logic...

    }

}

```

Declaring/Instantiating an Object from a Class

```
myClass myObj = new myClass("Hello"); //calls constructor
```

Use '.' to access object's methods and fields

```
int myResult = myObj.AddNumbers(5,10);
```

Static members don't require instantiation to use them

Instance members get copied each time a new object is created
 Static members belong to the entire class - there's only one of them

Example

```

class Wine{

    int Year;

    string Name;

    static int bottleCount = 0; //static member (only ONE PER CLASS)

    public Wine(string s, int y) {

```

```
        Name = s;

        Year = y;

        bottleCount += 1;

    }

}
```

Differentiating between Object Instance and Class

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace DefiningAClass

{

    class MyClass

    {

        int myInteger;

        string myMessage;

        public static int myStaticInt = 100;

        public int myFunction()

        {

            return myInteger;

        }

    }

}
```

```

    }

    public MyClass()

    {

        myInteger = 50;

        myMessage = "This is a string";

    }

}

class Program

{

    static void Main(string[] args)

    {

        MyClass myC = new MyClass();

        Console.WriteLine("Calling myFunction: {0}",
myC.myFunction());

        Console.WriteLine("Using a static member: {0}",
MyClass.myStaticInt);

        Console.ReadLine();

    }

}

}

```

Class Access Modifiers

```
class myClassName {  
  
    private            int someInteger;  
  
    public             string myMessage;  
  
    protected         void myfunction ();  
  
}
```

- private class member can only be accessed within the class itself
- public class member can be accessed by any other object
- protected class member can only be accessed within this class or any subclass (class that inherits from this class)
- internal class member can be accessed by any class in the same assembly (package of code like a library or other program)

Defining Properties

- Properties are like data fields, but have logic behind them
- From outside look like any other member variable
 - But they act like a member function
- Defined like a field, with “get” and “set” code added
- can use access modifiers like field and methods can
- Hides class logic, selectively allow what public sees
- Why Use?
 - Properties can be calculated on the fly
 - Provide complete control over the data field access
 - Logic is hidden from the object’s consumer
 - Properties promote the idea of encapsulation

Naming Conventions

```
Private string _firstName;  
  
Public string FirstName  
  
{  
  
    get { return _firstName; }  
  
    set { _firstName = value; }  
}
```

```
}
```

Example

```
class myExample {  
  
    private int someVal; //private data field  
  
    public int CurrentValue {  
  
        get { return someVal;}  
  
        set { someVal = value; } // value is implicit  
  
    }  
  
}
```

```
myExample myEX = new myExample();  
  
int i = myEX.CurrentValue;           // triggers the get logic  
  
myEX.CurrentValue = 5;               // triggers the set logic
```

Automatic properties:

```
class myExample {  
  
    private int someVal; //private data field  
  
    public int CurrentValue {  
  
        get { return someVal;}  
  
        set { someVal = value; } // value is implicit  
  
    }  
  
}
```



```
class myExample {  
  
    public int CurrentValue { get; set; }  
  
}
```

Read-only and Write-only properties

Read-Only

```
class myExample {  
  
    private int someVal; //private data field  
  
    public int CurrentValue {  
  
        get { return someVal;}  
  
    }  
  
}
```

Write-Only

```
class myExample {  
  
    private int someVal; //private data field  
  
    public int CurrentValue {  
  
        set { someVal = value; } // value is implicit  
  
    }  
  
}
```

Hide 'how we get the price' from the user

```

class Wine {

    public decimal Price; {

        get { return wholesalePrice * retailMarkup;}

    }

    private decimal wholesalePrice;

    private decimal retailMarkup;

}

```

Properties: Keyword 'value'

- myClass.Price always has a value.
- **value** holds this value

```

public decimal Price

{

    get

    {

        return wholesalePrice * retailMarkup;

    }

    set

    {

        wholesalePrice = value;

    }

}

```

Value Type Compared to Reference Types

- Differ in how they are handled in memory



Passing Value Types

- Passing value types to a function, a copy is made
- Original stays the same

```
int myInt = 10;
```

```
testFunc1(myInt);
```

```
testFunc1(int arg1)
```

```
{
```

```
    //change arg1
```

```
    return
```

```
}
```

Passing Pointer Types

- Pointers pass by reference
- Original is changed along with the reference in the function

```
Point p = new Point();
```

```
p.x = 10;
```

```
p.y = 10;
```

```
testFunc2(p); // p.x = 10
```

```
static void testFunc2(Point pt)
```

```
{
```

```
    pt.x += 10; //change pt by 10
```

```
}
```

```
// p.x = 20
```

Chapter 06: Collections

Arrays

- Fixed in size, Contiguous memory
- Very efficient
- Elements all the same type

Single Dimensional Array

```
int oneValue;
```

```
oneValue = 100;
```

```
int [] manyValues = new int[4];
```

```
int[] manyValues = { 100,200,300,400};
```

```
manyValues[100] = 500; //this will cause an error
```

Multi-Dimensional Arrays

```
int [,] multiDimArray = new int[3,3];  
multiDimArray[0,1] = 10;
```

Create and initialize

```
int[,] multiDimArray = new int[,] {  
  
    { 0,1,2 },  
  
    { 3,4,5 },  
  
    { 6,7,8 }  
  
};
```

Array Functions

GetLength()
Clear()
Sort()
Reverse()

ArrayLists

```
ArrayList myAL = new ArrayList();
```

```
ArrayList myAL = new ArrayList(capacity);
```

Adding/Removing

```
myAL.Add(object item)
```

```
myAL.Insert(int index, object item)
```

```
myAL.Remove(object item)
```

```
myAL.RemoveAt(int index)
```

```
myAL.Count
```

Indexing

```
myAL[0]
```

```
myAL[0] = "hello";
```

```
ArrayList myAL = new ArrayList();
```

```
myAL.Add("item 1");
```

```
myAL.Add("item 2");
```

```
myAL.Add("item 3");
```

```
myAL.Add("item 4");
```

Foreach

```
foreach ( object obj in myAL )
```

```
{
```

```
    Console.WriteLine("{0}",obj);
```

```
}
```

```
ArrayList myAL = new ArrayList();
```

```
myAL.Add("one");
```

```
myAL.Add(2);
```

```
myAL.Add("three");
```

```
myAL.Add(4);
```

Check to see if object is an integer, if so add to running count;

```
int result=0;
```

```
foreach (object obj in myAL)
```

```
{
```

```
    if (obj is int)
```

```
        result += (int)obj;
```

```
}
```

Stacks: LIFO

List array, but push-on, push-off

```
Stack myStack = new Stack();
```

```
myStack.Push("string 1");
```

```
myStack.Push("string 2");
```

```
object o = myStack.Pop(); // gets top value
```

```
object o = myStack.Peek(); // looks at top
```

```
int c = myStack.Count; // gets number of items in the stack
```

```
myStack.Clear(); // zeros the stack
```

Example:

```

Stack myStack = new Stack();

myStack.Push("item 1");
myStack.Push("item 2");
myStack.Push("item 3");
Console.WriteLine("{0} Items on the stack", myStack.Count);

// Have a peek at the top item
Console.WriteLine("{0}", myStack.Peek());

myStack.Pop(); // pops "item 3" off the top
// now "item 2" is the top item
Console.WriteLine("{0}", myStack.Peek());

myStack.Clear(); // get rid of everything
Console.WriteLine("{0} Items on the stack", myStack.Count);

```

Queues: FIFO

```

Queue myQ = new Queue();

myQ.Enqueue("string 1");

myQ.Enqueue("string 2");

object o = myQ.Dequeue(); // gets first item

object o = myQ.Peek();

int c = myQ.Count;

using System.Collections;

using System.Collections.Generic;

```

Example:

```

Queue myQ = new Queue();

myQ.Enqueue("item 1");
myQ.Enqueue("item 2");
myQ.Enqueue("item 3");
myQ.Enqueue("item 4");

Console.WriteLine("There are {0} items in the Queue", myQ.Count);

while (myQ.Count > 0)
{
    string str = (string)myQ.Dequeue();
    Console.WriteLine("Dequeueing object {0}", str);
}

```



```
int counter = myQ.Count;
Console.WriteLine("Dequeueing object {0}", counter);

Console.ReadLine();
```

Dictionaries: (Hashtable/Associative Array)

- Associate a key with a given value
- Lookup tables
- No sense of order

Hashtable

```
Hashtable myHT = new Hashtable();

myHT.Add("SEA", "Seattle Tacoma Airport");

myHT.Add("SFO", "San Francisco Airport");

myHT["IAD"] = "Washington Dulles Airport";

myHT.Remove("SFO");

int c = myHT.Count;

bool b = myHT.ContainsKey("SFO");

bool b = myHT.ContainsValue("San Francisco Airport");
```

Example:

```
Hashtable myHT = new Hashtable();

myHT.Add("SFO", "San Francisco Airport");

myHT.Add("SEA", "Seattle Tacoma Airport");

myHT["IAD"] = "Washington Dulles Airport";
```

```

myHT["SEA"]);        Console.WriteLine("Value for key {0} is {1}", "SEA",

                        Console.WriteLine("There are {0} items", myHT.Count);

                        //myHT.Remove("SFO");

                        if (myHT.ContainsKey("SFO")) {

myHT["SFO"]);        Console.WriteLine("Value for key {0} is {1}", "SFO",

```

Chapter 07

Method Overloading

- Duplicate method name with different arguments
-

Correct

```

class Wine {

    public int Year;

    public string Name;

    public decimal price;

    public Wine(string s){

        Name = s;

    }

    public Wine(strins s, int y){

```

```
        Name = s;

        Year;

    }

}
```

Error

```
class Example {

    public int function(int x){ }

    public float function1(int x) { }

}
```

Example:

```
class Program

{

    class Wine

    {

        public int Year;

        public string Name;

        public decimal price;

        public Wine(string s)

        {
```

```

        Name = s;

    }

    public Wine(string s, int y)

    {

        Name = s;

        Year = y;

    }

    public Wine(string s, decimal p, int y)

    {

        Name = s;

        price = p;

        Year = y;

    }

}

static void Main(string[] args)

{

    Wine w1 = new Wine("Charles Shaw Merlot");

    Wine w2 = new Wine("Mark Ryan Dissident", 2004);

    Wine w3 = new Wine("Dom Perignon", 120.00m, 1994);


    Console.WriteLine("{0}", w1.Name);

    Console.WriteLine("{0} {1}", w2.Year, w2.Name);

    Console.WriteLine("{0} {1} {2}", w3.Year, w3.Name, w3.price);

```

```

        Console.ReadLine();
    }
}

```

Overriding Methods

change or augment the behavior of methods in classes, overriding their logic



- **virtual** - Tells compiler that this method can be overridden by derived classes

```
public virtual int myFunction() {}
```

- **override** - in the subclass, tells the compiler that this method is overriding the same named method in the base class

```
public override int myFunction() {}
```

- **base** - in the subclass, calls the base class' method

```
base.myFunction();
```

Example

```

namespace MethodOverriding
{
    class baseClass
    {
        public virtual void doSomething() // baseClass does nothing by
default

```

```

        {

            Console.WriteLine("This is the baseClass saying hi!");

        }

    }

    class subClass : baseClass

    {

        public override void doSomething() // subClass adds
        functionality to baseClass

        {

            base.doSomething();

            Console.WriteLine("This is the subClass saying hi!");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            baseClass obj1 = new subClass();

            obj1.doSomething();

            Console.ReadLine();

        }

    }

```

```
        }  
  
    }  
  
}
```

Abstract Classes

Cannot be instantiated by themselves - you must create a subclass (derived class) and instantiate that instead.

have abstract members that derived classes must override in order to provide functionality

```
public abstract class myClass {  
  
    public abstract void function(int arg1);  
  
}
```



Example

```
namespace AbstractClasses  
  
{  
  
    abstract class myBaseClass  
  
    {  
  
        public abstract int myMethod(int arg1, int arg2);  
  
    }  
  
}
```

```

class myDerivedClass : myBaseClass

{

    // MUST OVERRIDE BASE CLASS or build error occurs

    public override int myMethod(int arg1, int arg2)

    {

        return arg1 + arg2;

    }

}


class Program

{

    static void Main(string[] args)

    {

        /*

            myDerivedClass mdc = new myDerivedClass();

            int result = mdc.myMethod(5, 6);

            Console.WriteLine("{0}", result);

        */

        // Error Cannot create an instance of the abstract class

        // myBaseClass mbc = new myBaseClass();

        Console.ReadLine();
    }
}

```



```
        }  
    }  
}
```

Sealed Classes and Methods

- Sealed classes are the opposite of abstract classes
- Abstract classes FORCE you to derive a subclass in order to use them
- Sealed classes PREVENT further subclasses from being derived
- Individual methods can also be marked as sealed, which prevents them from being overridden in subclasses
 - far more common to seal classes than methods

```
public sealed class myClass {  
  
}  
  
public class mySubClass : myClass {  
  
    // ERROR  
  
}
```

Example

```
namespace SealedClasses  
{  
  
    sealed class myExampleClass  
    {  
  
        public static string myMethod(int arg1)  
        {  
  
            return String.Format("You sent me the number {0}", arg1);  
  
        }  
    }  
}
```

```

    }

}

class mySubClass : myExampleClass

{

    //Error 1 'SealedClasses.mySubClass': cannot derive from sealed
type 'SealedClasses.myExampleClass' D:\DEVL\C#\Solutions\Exercise
Files\Ch7\SealedClasses\SealedClasses\Program.cs 16 11 SealedClasses

}

class Program

{

    static void Main(string[] args)

    {

    }

}

}

```

Defining STRUCTS

- Structs are similar to classes, but don't support inheritance
- Structs are value types, while classes are reference types
- You can't initialize the fields of a struct inside the definition
 - `this = 1; // can't do this`
- Structs are usually small and simple
- Structs can have properties and methods, just like a class
- **Good when you don't want all the overhead of a class with overriding, etc**

Definition

```
public class Point {
```

```

    public int xCoord;

    public int yCoord;

    public Point (int x, int y){ // constructor function

        xCoord = x;

        yCoord = y;

    }

}

public struct Point {

    public int xCoord;

    public int yCoord;

    public Point (int x, int y){ // constructor function

        xCoord = x;

        yCoord = y;

    }

}

```

Example

```

namespace UsingStructs

{

    public struct Point

    {

        private int xCoord;

```

```
private int yCoord;
```

```
public Point(int x, int y)
```

```
{
```

```
    xCoord = x;
```

```
    yCoord = y;
```

```
}
```

```
public int x
```

```
{
```

```
    get { return xCoord; }
```

```
    set { xCoord = value; }
```

```
}
```

```
public int y
```

```
{
```

```
    get { return xCoord; }
```

```
    set { xCoord = value; }
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```

        static void Main(string[] args)

        {

            Point p1 = new Point(100, 100);


            Point p2 = new Point();

            p2.x = 50;

            p2.y = 75;

        }

    }
}

```

Interfaces

- Provide a way of grouping a set of common behaviors in a single place where more than one class can use them
- Like a class, but provides a specification rather than an implementation. Alternative to abstract classes
- Classes can implement multiple interfaces
- Classes are THINGS
- Interfaces are BEHAVIORS
- Class can advertise to the world “I communicate this way!!”
- If many classes behave the same way, use interface to know how to reliably use them (predictable)

Definition

- No logic, just framework/contract

```

interface ISaveable {

    void SaveData();

}

interface IShrinkable{

    void CompressData();
}

```

```
}
```

```
class A : ISaveable, IShrinkable {  
  
    public void SaveData() {}  
  
    public void CompressData() {}  
  
}
```

```
class B : ISaveable, IShrinkable {  
  
    public void SaveData() {}  
  
    public void CompressData() {}  
  
}
```

```
class C : ISaveable, IShrinkable {  
  
    public void SaveData() {}  
  
    public void CompressData() {}  
  
}
```

```
class D : ISaveable{ // OK not to implement all interfaces  
  
    public void SaveData() {}  
  
}
```

Example

```
namespace UsingInterfaces  
  
{  
  
    interface ITalkative  
  
    {
```

```
void SayHello();

void SayGoodBye();

}

class myExampleClass : ITalkative

{

    public myExampleClass() // constuctor

    {

    }

    public void SayHello() // MUST impement

    {

        Console.WriteLine("Saying hello!");

    }

    public void SayGoodBye() // MUST impement

    {

        Console.WriteLine("Saying goodbye!");

    }

}

class Program
```

```

{

    static void Main(string[] args)

    {

        myExampleClass myEC = new myExampleClass();

        myEC.SayHello();

        myEC.SayGoodBye();

        Console.ReadLine();

    }

}

```

Chapter 08

C# Exceptions

- Exception is an error that happens during your program's execution
- Exceptions are **raised**, or **thrown**, when errors are encountered
- Program flow is interrupted, and control is transferred to the nearest exception handler - the handler **catches** the exception
- If there is no exception handler to process the exception, and the exception is fatal, then the program terminates with an error code
- Exceptions don't just come from .NET - you can build your own exceptions and then raise them yourself with the **throw** instruction.

```
x = 1;
```

```
y = 0;
```

```
result = x / y; //Raises DivideByZeroException
```


Definition

```
int x = 10, y = 0;

int result;

try { // code attempts an operation

    result = x / y;

}

catch { // exceptions are handled here

    Console.WriteLine("An error happened!");

}

finally { // this code is always executed

}

}
```

Example

```
namespace UsingExceptions

{

    class Program

    {

        static void Main(string[] args)

        {

            int x = 10;
```

```
int y = 0;

int result;

try

{

    result = x / y;

    Console.WriteLine("The result is: {0}", result);

}

catch

{

    // 10 / 0 = DevideByZeroException...so this TypeCode runs

    Console.WriteLine("An error occurred! Better check the
code!");

}

finally

{

    Console.WriteLine("Just proving that this code always
runs.");

}

Console.ReadLine();

}

}
```

Exception Object

- Base class is System.Exception
- System.ApplicationException
- <http://msdn.microsoft.com/en-us/library/c18k6c59%28v=vs.80%29.aspx>

```
try { }

catch (System.DivideByZeroException) {}

...

catch{System.ArithmeticException} {}
```

Example

```
namespace ExceptionObject

{

    class Program

    {

        static void Main(string[] args)

        {

            int x = 10;

            int y = 0;

            int result;

            try

            {

                result = x / y;
```

```

        Console.WriteLine("The result is: {0}", result);

    }

    // place catches from SPECIFIC to GENERAL

    //                                SubClass            Class

handler    catch (System.DivideByZeroException e) // e is passed into catch

    {

        //subclass of ArithmeticException so MUST BE FIRST

        Console.WriteLine("Whoops! You tried to divide by zero!");

message    Console.WriteLine(e.Message); // accses the actual error

    }

handler    catch (System.ArithmeticException e) // e is passed into catch

    {

        Console.WriteLine("Whoops! You tried to divide by zero!");

message    Console.WriteLine(e.Message); // accses the actual error

    }

    finally

    {

runs.");    Console.WriteLine("Just proving that this code always

    }

    Console.ReadLine();

```

```

        }

    }

}

```

Creating Custom Exceptions

By subclassing the `System.Exception` object, you can create your own
To raise an exception within your code, use the **throw** instruction

```

throw new System.Exception();

throw new MyCustomException();

```

Example

```

namespace CustomExceptions

{

    // SEE OVERRIDING METHODS ( REPLACING METHODS )

    // Base Class is System.Exception(string)

    public class NoJoesException : Exception

    {

        public NoJoesException()

        {

            : base("We don't allow no Joes in here!")

            /*

            * base class method is called and passes string to the
CONSTRUCTOR

            * ( Exception(string) )

            * MUST use constructor to pass error string

```

```

        */

    {

        /*

        * this.Message = "No Joes";

        *

        * Error      1      Property or indexer 'System.Exception.Message'
cannot be assigned to -- it is read only      D:\DEVL\C#\Solutions\Exercise
Files\Ch8\CustomExceptions\CustomExceptions\Program.cs      14      13
CustomExceptions

        */

        this.HelpLink = "http://www.joemarini.com/";

    }

}

class Program

{

    static string GetName()

    {

        string s = Console.ReadLine();

        if (s.Contains("Joe"))

            throw new NoJoesException();

        return s;

    }

}

static void Main(string[] args)

```

```
{

    string theName;

    try

    {

        theName = GetName();

        Console.WriteLine("Hello {0}", theName);

    }

    catch (NoJoesException nje)

    {

        Console.WriteLine(nje.Message);

        Console.WriteLine("For help, visit: {0}", nje.HelpLink);

    }

    finally

    {

        Console.WriteLine("Have a nice day!");

    }

    Console.ReadLine();

}

}
```

ReThrowing Exceptions

throw

```
try{}

catch{

    //handle then...

    throw; // throw an error;

}
```

Example

```
namespace RethrowingExceptions

{

    class Program

    {

        static void DoSomeMath()

        {

            int x = 10, y = 0;

            int result;

            try

            {

                result = x / y;

                Console.WriteLine("Result is {0}", result);

            }

            catch { }

        }

    }

}
```



```

    }

    catch // catches error, prints message, then throws other
ExceptionHandlers

    {

        Console.WriteLine("Error in DoSomeMath()");

        // throw; take same exception I'm handling,

        // then re-throw it to be caught by other Handlers

        throw new ArithmeticException();

    }

}

static void Main(string[] args)

{

    try

    {

        DoSomeMath();

    }

    // DoSomeMath throws ArithmeticException which is caught here...

    catch (ArithmeticException e)

    {

        Console.WriteLine("Hmm, there was an error in there, be
careful!");

        // e is a long message, so is often replaced by custom
message

        Console.WriteLine(e);

```

```
}  
  
Console.ReadLine();  
  
}  
  
}  
  
}
```

Chapter 09

Streams and Files

Streams

move data in or out of a file, across the network, over the Internet, etc. the data has to be **streamed**



- Streams can be readable, writeable, or both
- Streams can also be seekable - in other words, some types of streams can support a current position that can be queried or modified
- All streams descend from the **System.IO.Stream** abstract base class
- <http://msdn.microsoft.com/en-us/library/ms229335%28v=vs.80%29.aspx>
- MUST use subclass to provide implementation



File and Directory Classes

- **File** - simplified, high-level functions for working with files
- **FileInfo** - More detailed, lower-level functions that give more control
- **FileStream** - Exposes a Stream object around a file, with very detailed control.

Working with Directories

- **Directory** - Provides simplified, high-level functions for working with folders
- **DirectoryInfo** - More detailed, lower-level folder functions that give more control

File Object: Working with Files

- Most methods are static

`File.Exists`

`File.Create`

`File.Copy`

`File.Move`

`File.Delete`

Example

```
namespace Existing
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            bool fileExists = false;
```

```
            string thePath =  
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

```
            string theFile = thePath + @"testfile.txt";
```

```
            fileExists = File.Exists(theFile);
```

```
            if (fileExists)
```

```
{

    Console.WriteLine("The file exists");

}

else

{

    Console.WriteLine("The file does not exist, creating it");

    File.Create(theFile);

}

if (fileExists)

{

    Console.WriteLine("It was created on {0}",
File.GetCreationTime(theFile));

    Console.WriteLine("It was last accessed on {0}",
File.GetLastAccessTime(theFile));

    Console.WriteLine("Moving the file...");

    File.Move(theFile, thePath + @"\newfile.txt");

}

Console.ReadLine();

}

}
```

Working with Directories(Folders)

- Similar to working with files, except that also contain other files and directories

`Directory.Exists`

`Directory.CreateDirectory`

`Directory.Delete`

`Directory.GetCurrentDirectory`

`Directory.GetFiles`

`Directory.GetDirectories`

`Directory.Move`

Example

```
namespace ExistingDir

{

    class Program

    {

        static void Main(string[] args)

        {

            string thePath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            // Check to see if a directory exists

            bool dirExists = Directory.Exists(thePath);

            if (dirExists)

                Console.WriteLine("The directory exists.");

        }

    }

}
```

```
else

    Console.WriteLine("The directory does not exist.");

Console.WriteLine();

// Write out the names of the files in the directory

string[] files = Directory.GetFiles(thePath);

foreach (string s in files)

{

    Console.WriteLine("Found file: " + s);

}

Console.WriteLine();

// Get information about each fixed disk drive

Console.WriteLine("Drive Information:");

foreach (DriveInfo d in DriveInfo.GetDrives())

{

    if (d.DriveType == DriveType.Fixed)

    {

        Console.WriteLine("Drive Name: {0}", d.Name);

        Console.WriteLine("Free Space: {0}", d.TotalFreeSpace);

        Console.WriteLine("Drive Type: {0}", d.DriveType);

        Console.WriteLine();

    }

}
```

```

    }

    Console.ReadLine();

}

}

}

```

The Path Helper Class

System.IO.Path provides collection of utilities for path operations
Path has a predictable structure:

```
Drive:\Path\filename.extension
```

Example

```

namespace PathOperations

{

    class Program

    {

        static void Main(string[] args)

        {

            // get the path to the documents folder from the Environment

class

            string thePath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            thePath += @"\testfile.txt";

            // Exercise the Path class methods

```

```

        Console.WriteLine("The directory name is {0}",
Path.GetDirectoryName(thePath));

        Console.WriteLine("The file name is {0}",
Path.GetFileName(thePath));

        Console.WriteLine("File name without extension is {0}",
Path.GetFileNameWithoutExtension(thePath));

        Console.WriteLine("Random file name for path is {0}",
Path.GetRandomFileName());

        Console.ReadLine();

    }

}

```

Writing and Reading Data

Writing data:

`File.AppendAllText` - adds string/string[] to end of existing file

`File.WriteAllBytes` - Bytes[]

`File.WriteAllLines` - Strings[]

`File.WriteAllText` - string/string[] replace existing file

Reading Data

`File.ReadAllText` - reads all file as bytes

`File.ReadLines`

`File.ReadAllLines`

`File.ReadAllText`

Example

```
namespace ReadingWritingData
```



```
{

    class Program

    {

        static void Main(string[] args)

        {

            // create a path to the My Documents folder and the file name

            string filePath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +

                                Path.DirectorySeparatorChar +
"examplefile.txt";

            // if the file doesn't exist, create it by using WriteAllText

            if (!File.Exists(filePath))

            {

                string content = "This is a text file." +
Environment.NewLine;

                Console.WriteLine("Creating the file...");

                File.WriteAllText(filePath, content);

            }

            // Use the AppendAllText method to add text to the content

            string addedText = "Text added to the file" +
Environment.NewLine;

            Console.WriteLine("Adding content to the file...");

            File.AppendAllText(filePath, addedText);

        }

    }

}
```

```

        Console.WriteLine();

        // Read the contents of the file

        Console.WriteLine("The current contents of the file are:");

        Console.WriteLine("-----");

        // Read the contents of the file using ReadAllText

        Console.WriteLine("Returns a STRING-----");

        string currentContent = File.ReadAllText(filePath);

        Console.Write(currentContent);

        Console.WriteLine();

        // Read the contents of the file using ReadAllLines

        Console.WriteLine("Returns an ARRAY of Strings-----");

        string[] contents = File.ReadAllLines(filePath);

        foreach (string s in contents)

        {

            Console.WriteLine(s);

        }

        Console.WriteLine();

        Console.ReadLine();

    }

```

```
    }  
  
}
```

Chapter 10

Variable Parameter Lists

Define a function that takes a variable number of argument parameters

```
// want to take variable number of arguments, but must use method overloading  
  
int addNumbers( int a, int b ) {  
  
    int total = 0;  
  
    total = a + b;  
  
    return total;  
  
}  
  
int addNumbers( params int[] nums ) {  
  
    int total = 0;  
  
    foreach (int x in nums) {  
  
        total += x;  
  
    }  
  
}
```

Example

```
namespace VariableParams  
  
{  
  
    class Program
```

```

{

    static void Main(string[] args)

    {

        int result;

        result = addNumbers(4, 6, 8);

        Console.WriteLine("result is {0}", result);

        result = addNumbers(12, 20, 31, 5, 7, 9);

        Console.WriteLine("result is {0}", result);

        Console.ReadLine();

    }

    // all other passed arguments must go BEFORE the array[] of arguments

// params must be LAST parameter

    static int addNumbers(params int[] nums)

    {

        int total = 0;

        foreach (int x in nums)

        {

            total += x;

```

```

    }

    return total;

}

}

```

Function Parameter Modifiers

REF

- **ref** causes a parameter to be passed by reference

```

void myFunction(int param1)

{

    param1 += 10; // doesn't change x below

}

```

```

void test()

{

    int x = 10;

    myFunction(x);

}

```

```

void myFunction(ref int param1)

{

    param1 += 10; // doesn't change x below

```

```
}
```

```
void test()
```

```
{
```

```
    int x = 10;
```

```
    myFunction(ref x); // x will now be 20
```

```
}
```

OUT

- **out** allows a return value to be passed back via a parameter
- passes back MORE THAN ONE result
- pass them in, they come out also

```
void SqrAndRoot( double num, out double sq, out double sqrt)
```

```
{
```

```
    sq = num * num;
```

```
    sqrt = Math.Sqrt(num);
```

```
}
```

```
void test()
```

```
{
```

```
    double x=9.0, theSquare, theRoot;
```

```
    SqrAndRoot(x, out theSquare, out theRoot);
```

```
}
```

Example

```
namespace FuncParamModifiers  
{
```

```

class Program
{
    static void SquareAndRoot(double num, out double sq, out double sqrt)
    {
        sq = num * num;
        sqrt = Math.Sqrt(num);
    }

    static void Main(string[] args)
    {
        double n = 9.0;
        double theSquare, theRoot;

        SquareAndRoot(n, out theSquare, out theRoot);
        Console.WriteLine("The square of {0} is {1} and its square root is {2}", n, theSquare,
theRoot);

        Console.ReadLine();
    }
}

```

Optional and Named Parameters

Optional Parameters

```
void myFunction(int a=0, int b=1)
```

```
myFunction()    //a defaults to 0, b to 1
```

```
myFunction(10) //a is 10, b defaults to 1
```

Named Parameters

- Named Parameters MUST go last
- Out of order OK

```
void myFunction(int a, int b){}
```

```
myFunction(b:5, a:3) // passes name and value to the function, and out of order
OK
```

```
myFunction(3, b:7) // a is 3 here
```

```
myFunction(b:7, 3) // ERROR
```

Example

```

namespace NamedOptionalParams

{

    class Program

    {

        static void Main(string[] args)

        {

            // call the Named parameters example

            //myNamedParamExample(stateName: "Washington", zipCode: 98121,
cityName: "Seattle");

            //myNamedParamExample(cityName: "San Francisco", zipCode: 94109,
stateName: "California");

            //myNamedParamExample(zipCode: 94109, cityName: "New York",
stateName: "New York");


            // call the Optional parameters example

            myOptionalParamFunc(15);

            myOptionalParamFunc(10, 2.71828d);

            myOptionalParamFunc(10, 2.71828d, "a different string");

            // ERROR because no version that takes int, then string

            //myOptionalParamFunc(10, "a different string");


            // Now do both

            // skips the middle parameter my using int, <skipping>, then
named parameter

            myOptionalParamFunc(10, param3: "named and optional in same
call!");

```



```

        Console.ReadLine();

    }

    static void myOptionalParamFunc(int param1, double param2 = 3.14159d,
string param3 = "placeholder string")

    {

        // \n = newline

        // \t = tab in

        Console.WriteLine("Called with: \n\tparam1 {0}, \n\tparam2 {1},
\n\tparam3 {2}", param1, param2, param3);

    }


    static void myNamedParamExample(string cityName, string stateName,
int zipCode)

    {

        Console.WriteLine("Arguments passed: \n\tcityName is {0},
\n\tstateName is {1}, \n\tzipCode is {2}",

            cityName, stateName, zipCode);

    }

}

```

C# Preprocessor Directives

- Give instructions to the compiler PRIOR to compiling
- define or undefine a symbol

```
#define
```

```
#undef
```

- use logic to see if a symbol is defined

```
#if
```

```
#else
```

```
#elif
```

```
#endif
```

- Used for documentation of code
- Creates collapsible regions in IDE with optional message
- Doesn't change functionality

```
#region This is the main function
```

```
#endregion
```

Example

```
// defining symbols go before first token in file
```

```
// symbols hold NO value
```

```
#define DEBUGCODE
```

```
#define JOE
```

```
// token..
```

```
using System;
```

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

```
using System.Linq;
```

```
using System.Text;
```

```
namespace Preprocessor

{

    class Program

    {

        #region This is the main function

        static void Main(string[] args)

        {

            // only run if DEBUGCODE EXISTS

            #if DEBUGCODE

                Console.WriteLine("This is only in debug code");

            #else

                Console.WriteLine("This only gets written out in non-debug
code");

            #endif

            #if JOE

                Console.WriteLine("Joe was here!");

            #endif

            Console.ReadLine();

        }

    }

}
```

```
        #endregion

    }

}
```

C# Delegates

- C# delegates decouple the caller and the called function, like callbacks.
- Allows to use same 'NAME' to perform different functions
- why use?
- f = Square;
- f = Cube;

```
delegate int DelegateName (int i, string s);
```

```
DelegateName func = someFunction;
```

```
int result = func(3, "hello");
```

```
int someFunction(int i, string s)
```

```
{

    ...

}
```

Example

```
namespace UsingDelegates
```

```
{
```

```
    // before the class
```

```
    public delegate int NumberFunction(int x);
```

```
class Program

{

    static void Main(string[] args)

    {

        NumberFunction f = Square;

        Console.WriteLine("result of the delegate is {0}", f(5));

        // now change the delgate

        f = Cube;

        Console.WriteLine("result of the delegate is {0}", f(5));

        Console.ReadLine();

    }

    // Takes same values referenced in the Delegate

    static int Square(int num)

    {

        return num * num;

    }

    // Takes same values referenced in the Delegate

    static int Cube(int num)

    {

        return num * num * num;

    }

}
```

```
        }  
  
    }  
  
}
```

C# Events

Use Delegate

```
delegate void MyEventHandler (in x , string y);
```

```
class myClass{  
  
    public event MyEventHandler MyEvent;  
  
}
```

```
myClass obj = new myClass();
```

```
// listener/subscribing to an event
```

```
obj.MyEvent += handlerFunction; // START / ADD
```

```
obj.MyEvent -= handlerFunction; // STOP / REMOVE
```

```
// called on event
```

```
void handlerFunction(int x, string y) {...}
```

Example

```
namespace UsingEvents

{

    // have a delegate

    public delegate void myEventHandler(string newValue);

    class EventExample

    {

        // MEMBER

        private string theValue;

        public event myEventHandler valueChanged;

        public string Val

        {

            set

            {

                this.theValue = value;

                this.valueChanged(theValue);

            }

        }

    }

    class Program

    {

        static void Main(string[] args)
```

```

{

    EventExample myEvt = new EventExample();

    myEvt.valueChanged += new myEventHandler(myEvt_valueChanged);


    string str;

    do

    {

        str = Console.ReadLine();

        if (!str.Equals("exit"))

            myEvt.Val = str;

    } while (!str.Equals("exit"));

}

static void myEvt_valueChanged(string newValue)

{

    Console.WriteLine("The value changed to {0}", newValue);

}

}

}

```

Chapter 11

C# Garbage Collection

Example

```
namespace GarbageCollection

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Allocated memory is: {0}",
GC.GetTotalMemory(false));

            Console.WriteLine();

            Console.ReadLine();

            byte[] myArray = new byte[100000];

            Console.WriteLine("Allocated memory is: {0}",
GC.GetTotalMemory(false));

            Console.WriteLine();

            Console.ReadLine();

            GC.Collect();

            Console.WriteLine("Allocated memory is: {0}",
GC.GetTotalMemory(false));
```

```
Console.WriteLine();
```

```
Console.ReadLine();
```

```
}
```

```
}
```

```
}
```

Chapter 12

Debugging

BreakPoint (Click in Gutter for RED dot)





Debug Toolbar



- Next Breakpoint
- Step Into - Step into function and method calls
- Step Out - Execute until function exit to calling statement
- Step Over - current statement execute

Debug Class

- Allows to get Debug info without interfering with the programs execution
- Pipes info to OUTPUT window

```
using System.Diagnostics; // need this to use the Debug class
```

```
Debug.WriteLine("Created the file with content: {0}", (object)content);
```



Example

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Diagnostics; // need this to use the Debug class


namespace DebugAndTrace

{

    class Program

    {

        static void Main(string[] args)

        {

            // create a path to the My Documents folder and the file name

            string filePath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +

                                Path.DirectorySeparatorChar +
"examplefile.txt";

            // if the file doesn't exist, create it by using WriteAllText

            if (!File.Exists(filePath))

            {

                CreateFile(filePath);

            }

        }

    }

}
```

```

    }

    // Use the AppendAllText method to add text to the content

    WriteFileData(filePath, "Text to add to the file");

    WriteFileData(filePath, ""); //Causes an Assertion that is
handled

    // Read the contents of the file

    Console.WriteLine("The current contents of the file are:");

    Console.WriteLine("-----");

    ReadAndListFileContents(filePath);

    Console.ReadLine();

}

static void CreateFile(string filePath)

{

    string content = "This is a text file." + Environment.NewLine;

    Console.WriteLine("Creating the file...");

    File.WriteAllText(filePath, content);

    Debug.WriteLine("Created the file with content: {0}",
(object)content);

```

```
}
```

```
static void WriteFileData(string filePath, string content)
```

```
{
```

```
    Debug.Assert(content.Length > 0, "Tried to call WriteFileData with  
an empty string");
```

```
    Debug.Indent();
```

```
    Debug.WriteLine("Writing File Data");
```

```
    Debug.Unindent();
```

```
    string addedText = content + Environment.NewLine;
```

```
    Console.WriteLine("Adding content to the file...");
```

```
    File.AppendAllText(filePath, addedText);
```

```
    Console.WriteLine();
```

```
    Debug.Indent();
```

```
    Debug.WriteLine("File Data Written");
```

```
    Debug.Unindent();
```

```
}
```

```
static void ReadAndListFileContents(string filePath)
```

```
{
```

```
string[] contents = File.ReadAllLines(filePath);

Debug.WriteLineIf(contents.Length > 2, "The file has more than
two lines");

foreach (string s in contents)
{
    Console.WriteLine(s);
}

Console.WriteLine();
}
}
```

END