

# C#

Patric Boscolo  
Microsoft Academic Program



Microsoft

# C#

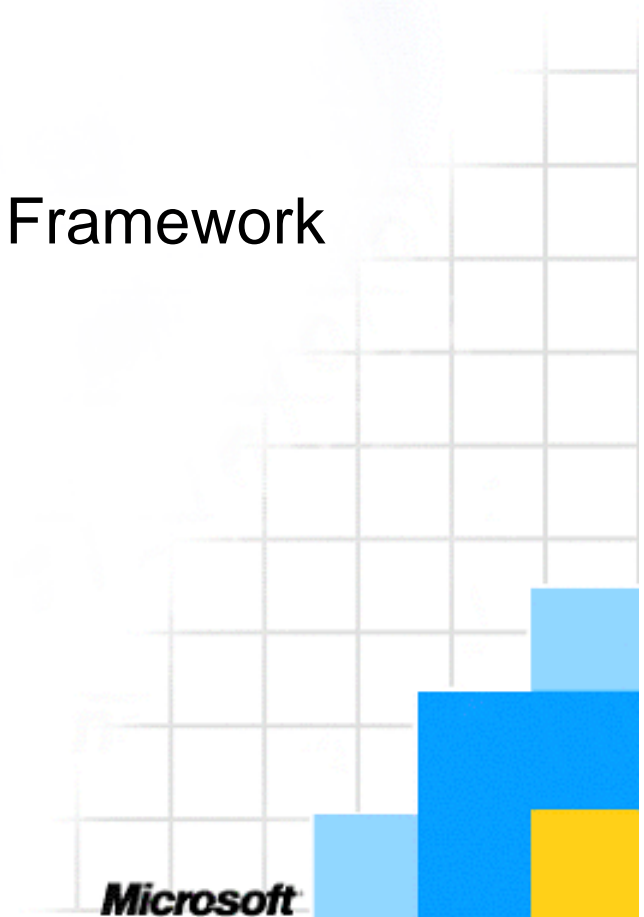
A Component-oriented Language for the  
Microsoft® .NET Framework



Microsoft

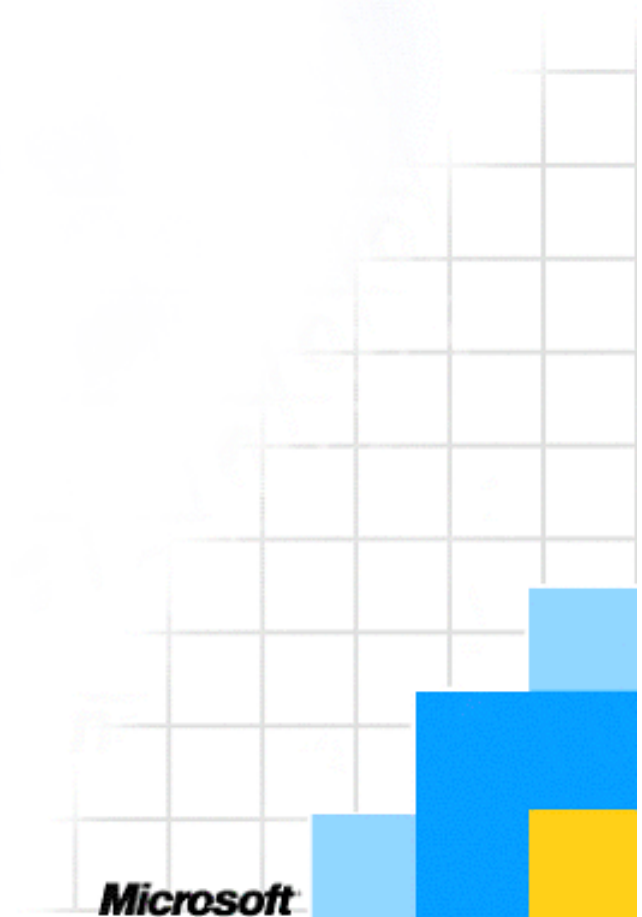
# Objectives

- Introduction to the C# Language
  - Syntax
  - Concepts
  - Technologies
- Overview of C# Tools and the .NET Framework



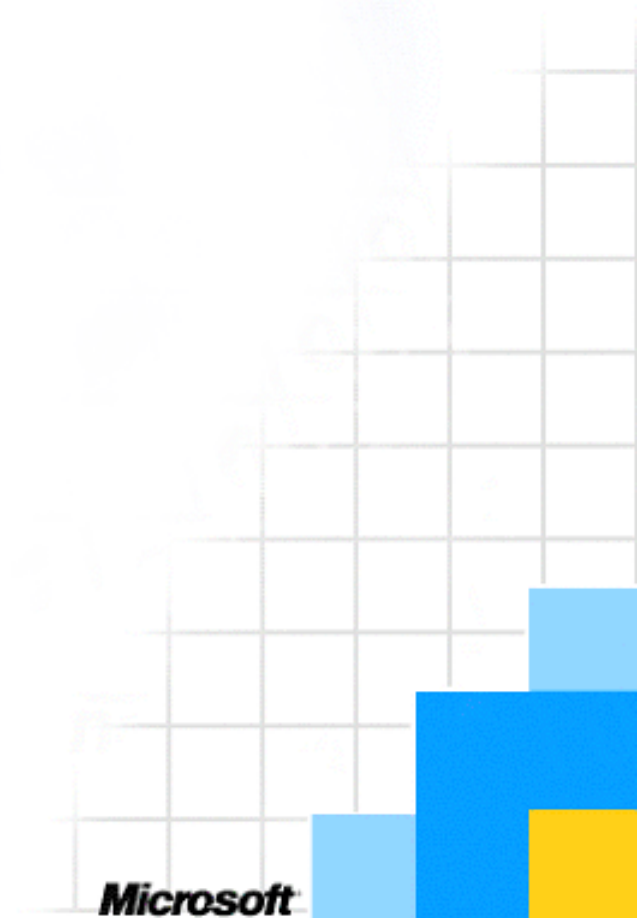
# Contents

- Section 1: C# Overview
- Section 2: Elements of C#
- Section 3: C# Tools
- Section 4: Putting It All Together
- Summary



# Section 1: C# Overview

- Component-oriented Systems
- Component Concepts of .NET
- Why C#?



# Component-oriented Systems

## ■ COM

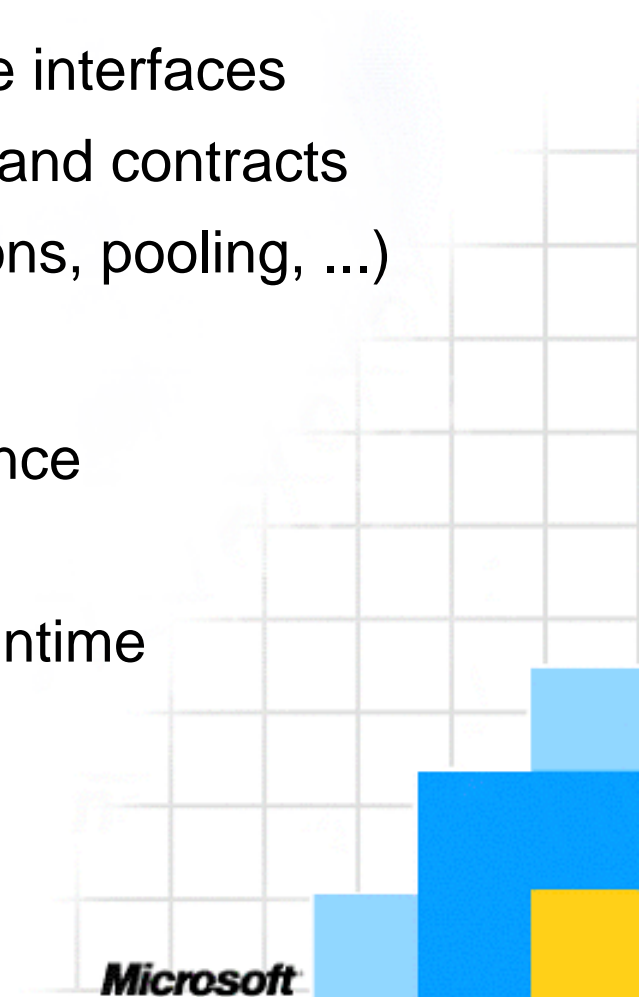
- Most successful component model in history
- Only viable commercial component platform
- Enables cross-organization integration and reuse

## ■ However:

- COM shows its age
  - DCOM does not function well over the Internet
  - More component-based systems, more "DLL Hell"
  - Even with maturing tools, still too difficult to implement
- COM is not truly language agnostic
  - Makes some assumptions of binary layout
  - Supports "least common denominator" type system only

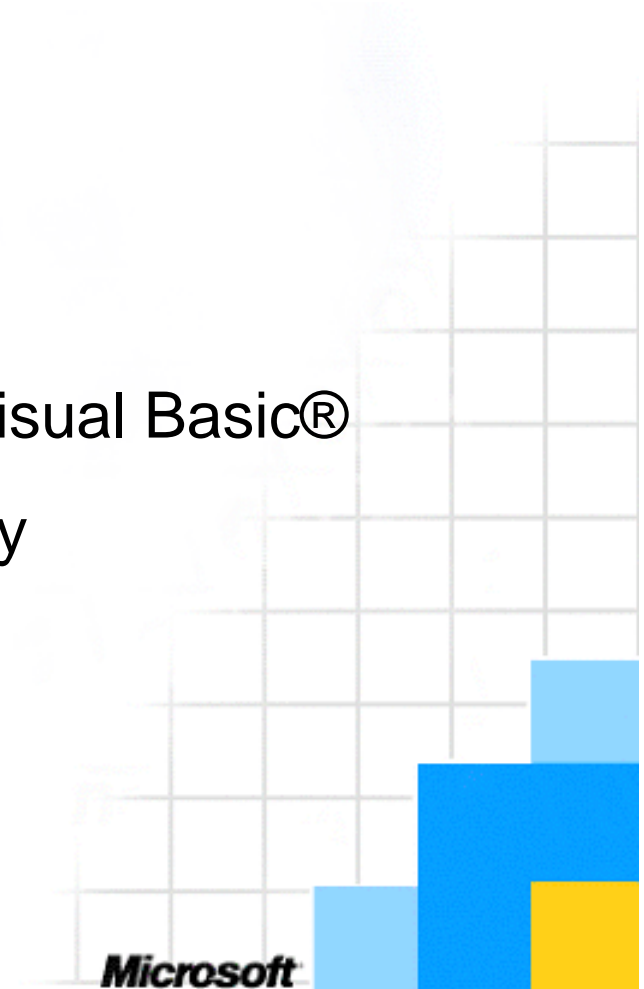
# ■ Component Concepts of .NET

- Take the best of COM+
  - Interfaces as abstract contracts
  - Components implement and expose interfaces
  - Dynamic exploration of capabilities and contracts
  - Attribute-driven behavior (transactions, pooling, ...)
- Add
  - True object-orientation and inheritance
  - Native events model
  - Cross-language type system and runtime
  - Extensibility at every level



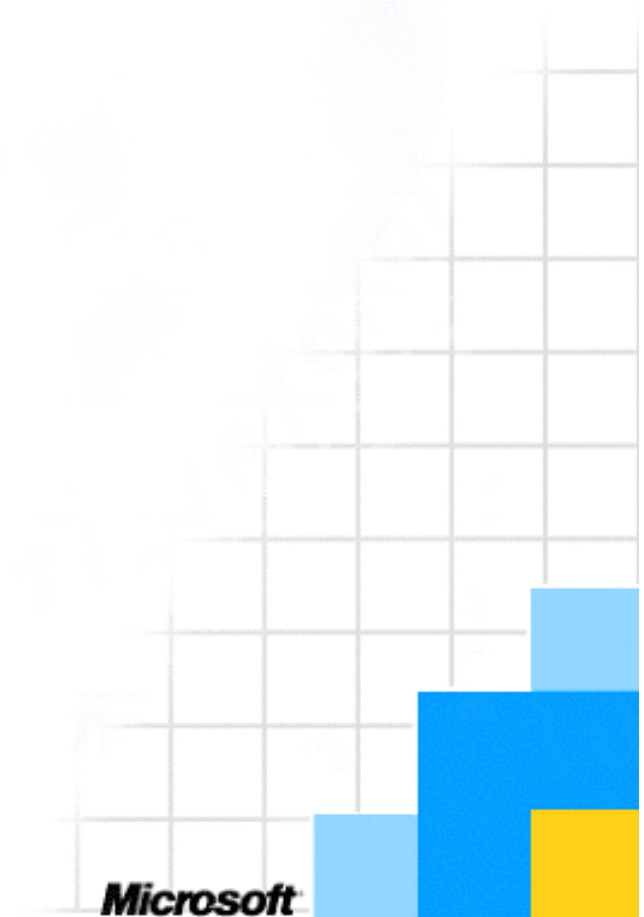
# Why C# ?

- First component-oriented language
  - Builds on COM+ experience
  - Native support for
    - Namespaces
    - Versioning
    - Attribute-driven development
- Power of C with ease of Microsoft Visual Basic®
- Minimal learning curve for everybody
- Much cleaner than C++
- More structured than Visual Basic
- More powerful than Java



## ■ Section 2: Elements of C#

- Shape and Structure
- Understanding the C# Type System
- Understanding the C# Language

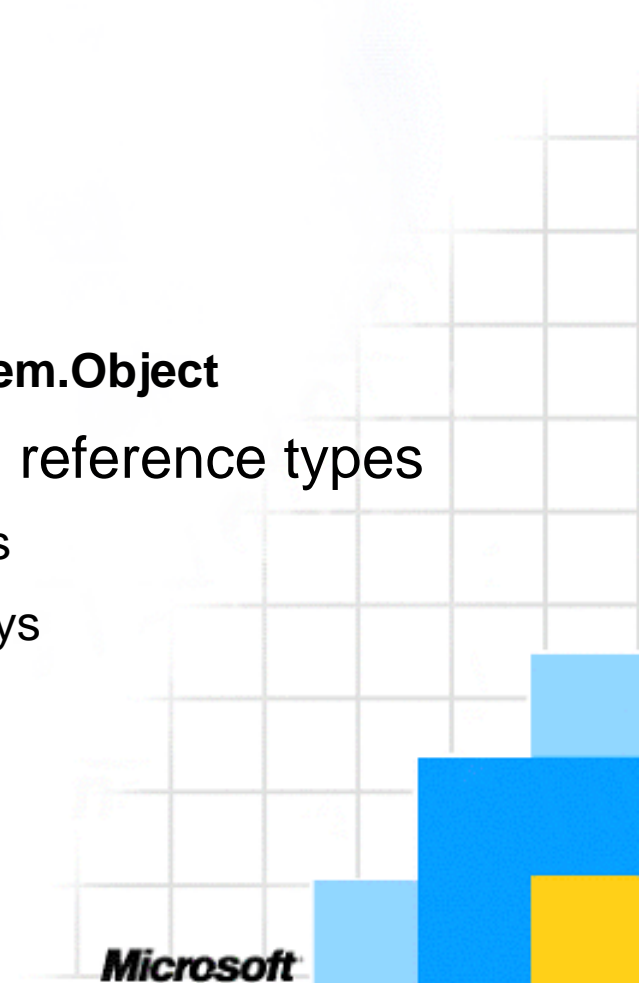


# ■ Shape and Structure

- No header files
- C# employs "definition at declaration" model
  - Much like Visual Basic, Pascal, Modula, Java
- Similar to C++ "inline" implementation
  - Without implication on code generation
- All code and declaration in one place
  - Keeps code consistent and maintainable
  - Much better to understand for team collaboration
  - Declaration accessible through metadata
- Conditional compilation but **no** macros

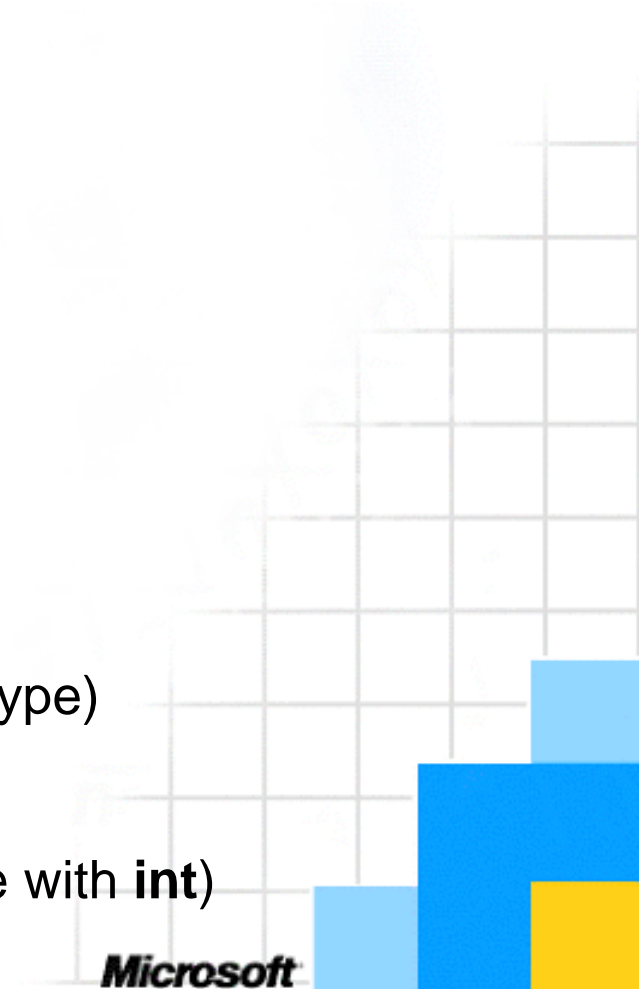
# Type System

- Builds on common type system of .NET Framework
- Native access to .NET type system
  - C# was born out of .NET
- Core concepts:
  - Everything is an object
    - Everything **implicitly** inherits from **System.Object**
  - Clear distinction between value and reference types
    - By-Value: Simple Types, Enums, Structs
    - By-Reference: Interfaces, Classes, Arrays



# Simple Types

- Integer Types
  - **byte**, **sbyte** (8bit), **short**, **ushort** (16bit)
  - **int**, **uint** (32bit), **long**, **ulong** (64bit)
- IEEE Floating Point Types
  - **float** (precision of 7 digits)
  - **double** (precision of 15–16 digits)
- Exact Numeric Type
  - **decimal** (28 significant digits)
- Character Types
  - **char** (single character)
  - **string** (rich functionality, by-reference type)
- Boolean Type
  - **bool** (distinct type, **not** interchangeable with **int**)

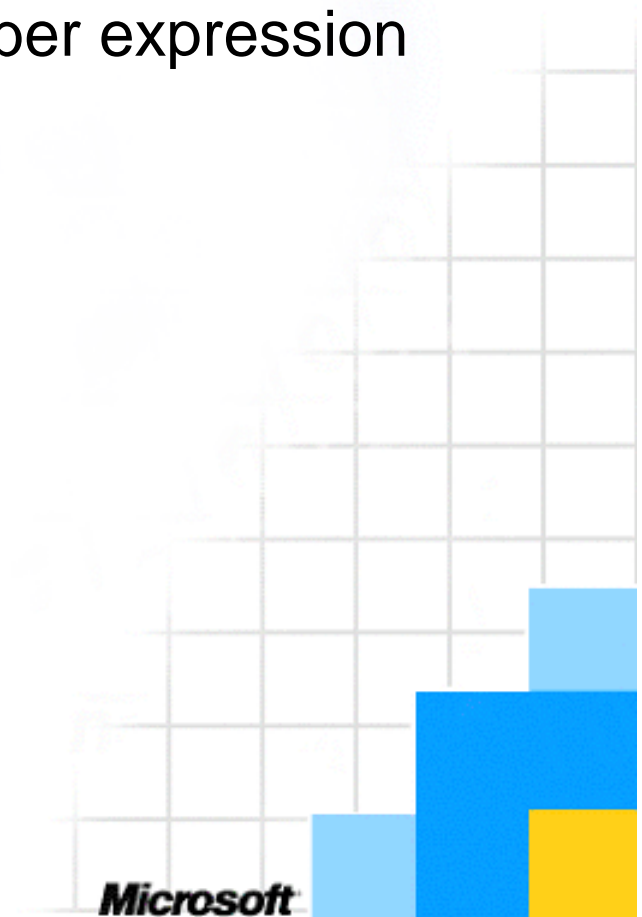


# Enums

- Named elements used instead of numbering options
- Strongly typed, no automatic conversion to **int**
- Better to use "Color.Blue" than number expression
  - More readable, better maintenance
  - Still as lightweight as a plain **int**

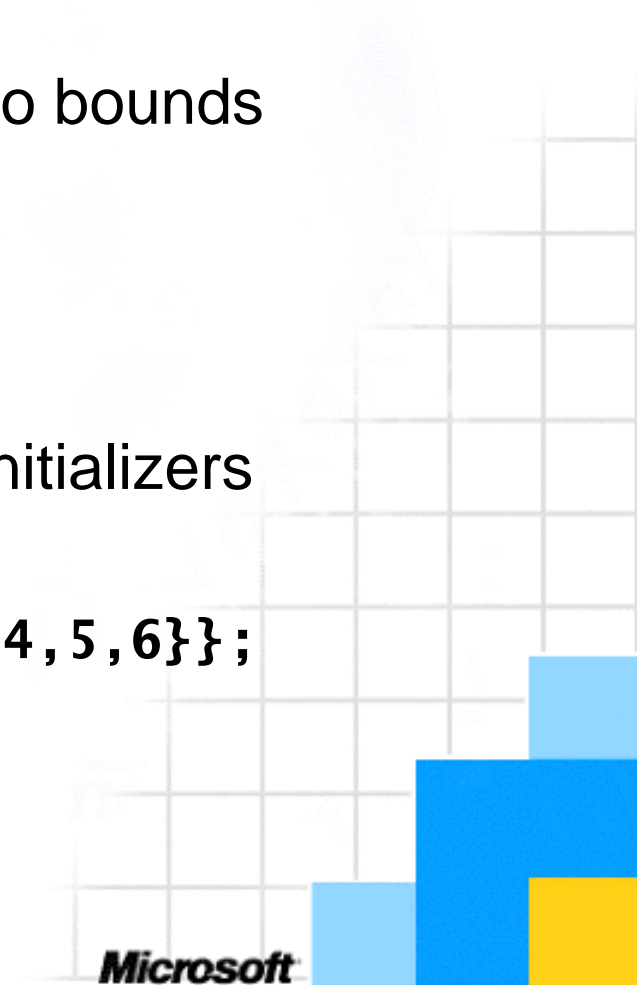
■ Example: 

```
enum Color
{
    Red,
    Green,
    Blue,
    Yellow
};
```



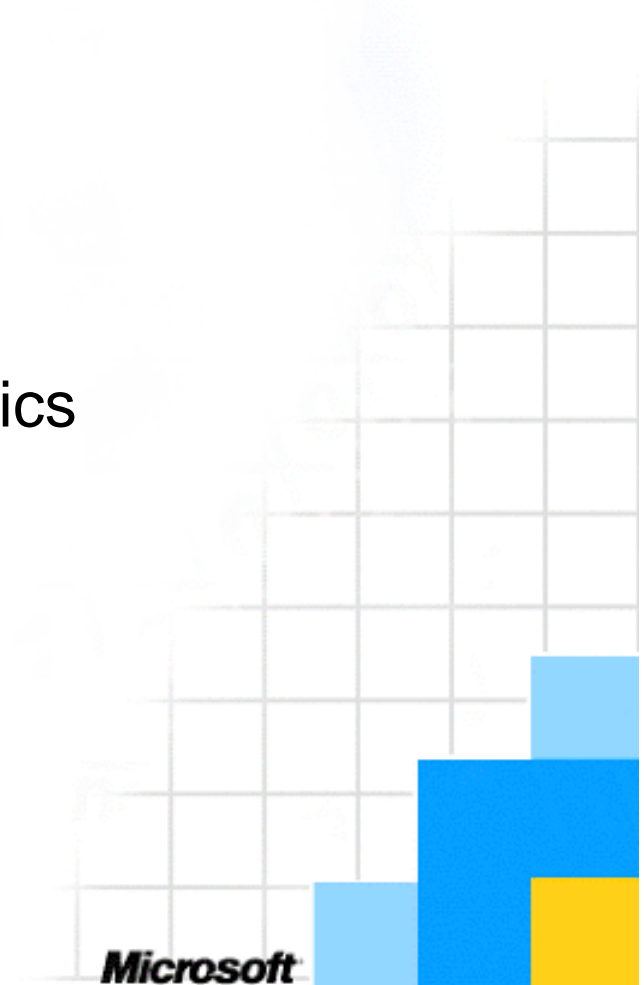
# Arrays

- Zero based, type bound
- Built on .NET **System.Array** class
- Declared with type and shape, but no bounds
  - `int[] SingleDim;`
  - `int[,] TwoDim;`
  - `int [][] Jagged;`
- Created using **new** with bounds or initializers
  - `SingleDim = new int[20];`
  - `TwoDim = new int[,]{{1,2,3},{4,5,6}};`
  - `Jagged = new int[1][];`  
`Jagged[0] = new int[]{1,2,3};`



# Namespaces

- Every definition must be contained in a namespace
  - Avoids name collisions
  - Enforces order
  - Makes API's easier to comprehend
- Can and should be nested
- Group classes and types by semantics
- Declared with keyword **namespace**
- Referenced with **using**



# ■ Interfaces

- Declarations of semantics contracts between parties
  - Enables component orientation
- Define structure and semantics for specific purposes
- Abstract definitions of methods and properties
- Support (multiple) inheritance

■ Example: 

```
interface IPersonAge
{
    int YearOfBirth {get; set;}
    int GetAgeToday();
}
```



# Classes

- Implementation of code and data
  - Represents semantic unit
- Implement interfaces
- Inherit from single base class
- Classes contain:
  - Fields: member variables
  - Properties: values accessed through **get/set** method pairs
  - Methods: functionality for object or class
  - Specials: events, indexers, delegates

```
public class Person :
    I PersonAge
{
    private int YOB;

    public Person()
    {
    }

    public int YearOfBi rth
    {
        get { return YOB; };
        set { YOB = value; };
    }

    public int GetAgeToday()
    {
        return Today() -
            YearOfBi rth
    };
}
```

# Structs

- Groups of data and code
  - Similar to classes, however:
    - No inheritance allowed
    - Always passed by value
  - Classes vs. Structs
    - Struct  $\Rightarrow$  Lightweight data container, value type
    - Class  $\Rightarrow$  Rich object with references, reference type
- C++ Developers!
  - Struct is not a class with everything being public

■ Example:

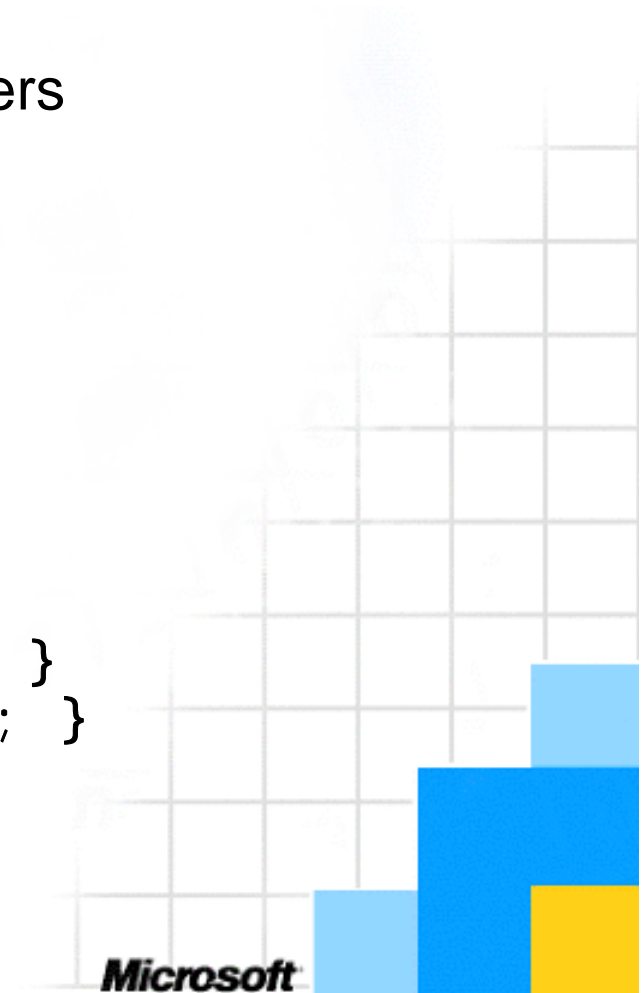
```
struct Point
{
    double X;
    double Y;
    void MoveBy(double dX, double dY)
    { X+=dX; Y+=dY; }
}
```

# ■ Properties

- Mix between fields and methods
- Use properties for:
  - Implementation of read-only members (by omitting **set**)
  - Validation on assignment
  - Calculated or composed values
  - Exposing values on interfaces

■ Example:

```
string Name
{
    get { return name; }
    set { name = value; }
}
```



# Indexers

- Consistent way to build containers
- Build on properties idea
- Allow indexed access to contained objects
- Index qualifier may be of any type

■ Example:

```
object this[string index]
{
    get { return Dict.Item(index); }
    set { Dict.Add(index, value); }
}
```

# Delegates

- Similar to function pointers found in C and C++
- Strongly typed, no type-cast confusion or errors
- Declaration creates typed method signature:
  - `delegate void Clicked(Element e, Point p);`
- Actual delegate is an instance of this type
  - `Clicked MyClickListener  
= new Clicked(obj.OnButtonClick);`
- Argument passed to delegate constructor:
  - Reference to object instance and method
  - Method must have the exact same signature
    - `void OnButtonClick(Element e, Point p) { ... };`

# Events

- Language-intrinsic event model
- All management done by C#
- Events are declared using a delegate type
- Declaration creates event source to bind to
  - `event Clicked OnClicked;`
- Event sinks bind to event source with delegates
  - Add handler:
    - `btnAction.OnClicked += MyClickListener;`
  - Remove handler:
    - `btnAction.OnClicked -= MyClickListener;`
- Event source triggers event sinks with single call
  - `OnClicked(this, PointerLocation);`

# Attributes

- Similar to attributes known from IDL
- Declarative access to functionality
- Extensible through custom attributes
- Allow code augmentation with:

- Hints for the runtime environment

```
[Transaction(TransactionOption.Required)]  
class MyBusinessComponent  
    : ServicedComponent { ... }
```

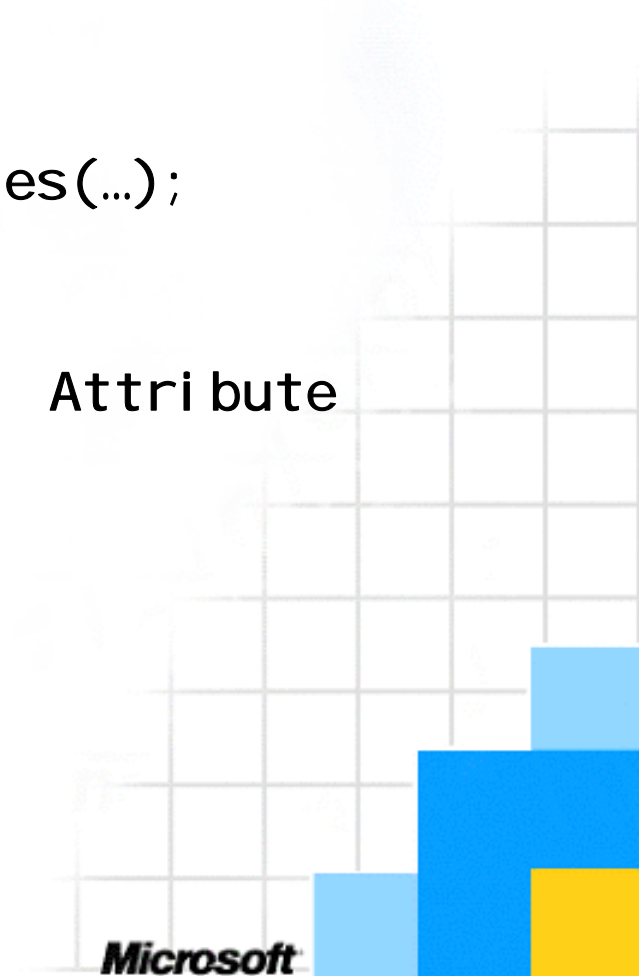
- Declarative semantics

```
[PersonFirstName] String Vorname;  
[PersonFirstName] String PrimarioName;
```

# ■ Creating Custom Attributes

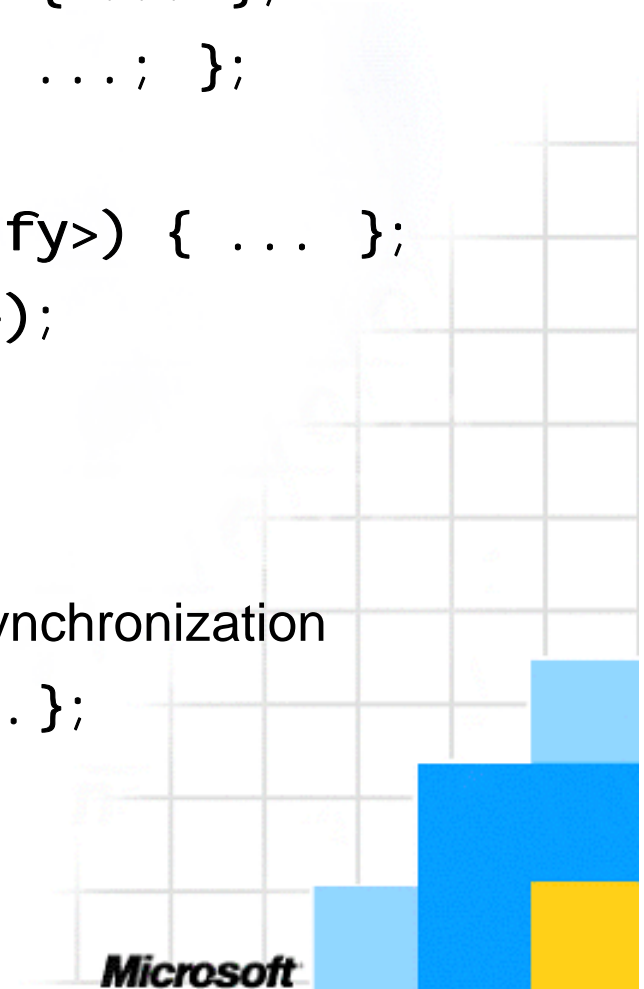
- Implement class with Attribute base class
- Automatically extend language (!)
- Explore through .NET reflection
  - 0. GetType(). GetCustomAttributes(...);
- Example:

```
class FirstNameAttribute : Attribute
{
    public FirstName()
    {
    }
}
```



# Statements

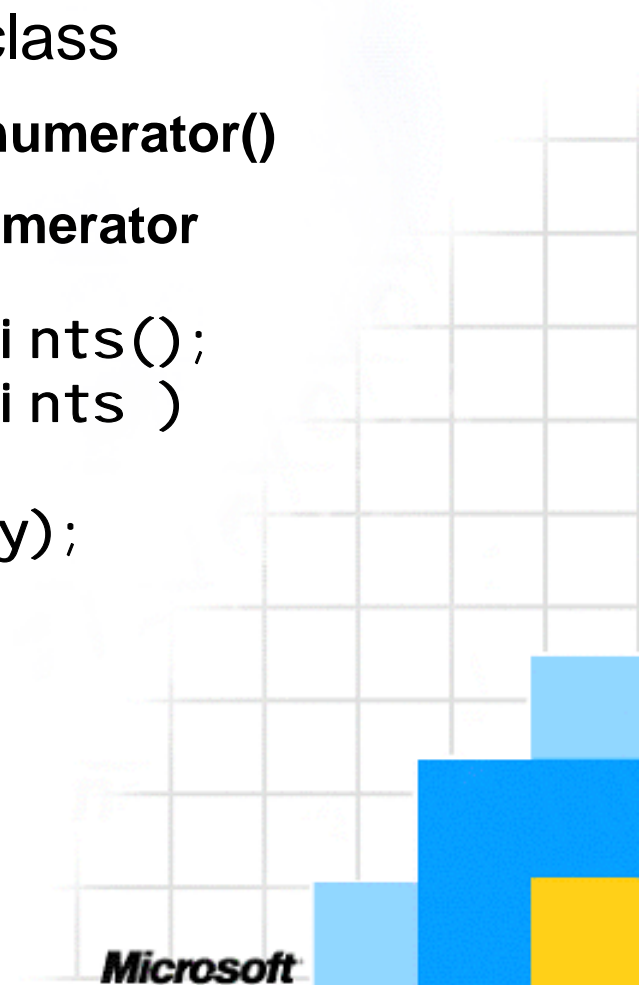
- Very C: Flow Control and Loops
  - `if (<bool expr>) { ... } else { ... };`
  - `switch(<var>) { case <const>: ...; };`
  - `while (<bool expr>) { ... };`
  - `for (<init>; <bool test>; <modify>) { ... };`
  - `do { ... } while (<bool expr>);`
- Very not C:
  - `lock(<object>){ ... };`
    - Language inherent critical section synchronization
  - `checked { ... }; unchecked { ... };`
    - Integer overflow protection



# ■ Collections Built-in: foreach

- Straightforward support for iterating over collections
  - Can be used for arrays and other collections
- Can also be used with any custom class
  - Implements **IEnumerable** with **GetEnumerator()**
  - Returning object implementing **IEnumerator**
- Example: 

```
Point[] Points = GetPoints();
foreach( Point p in Points )
{
    MyPen. MoveTo(p. x, p. y);
}
```



# Operators

## ■ Very C:

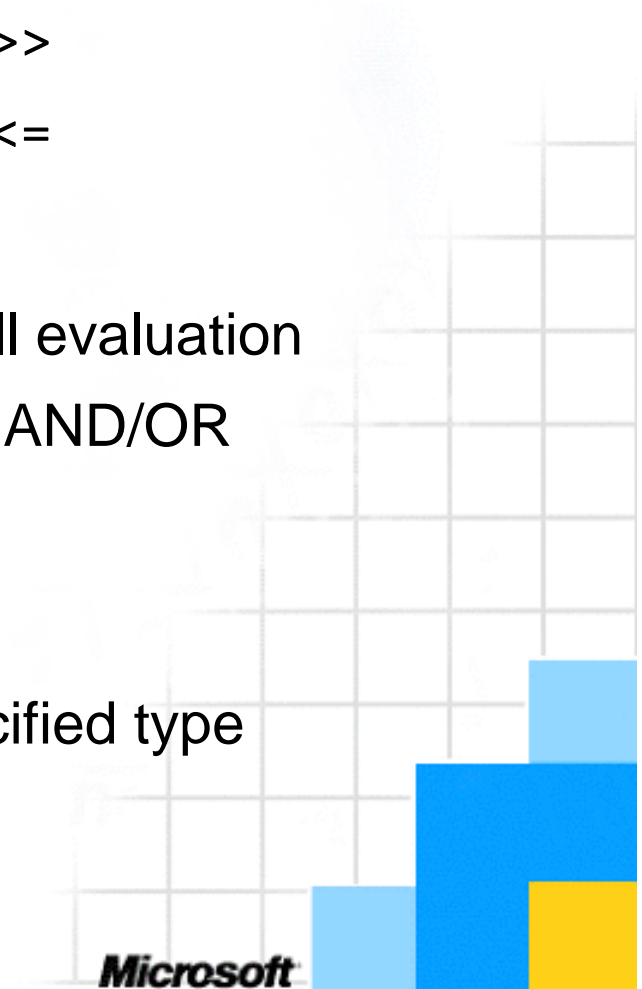
- Logical/conditional: `&&` `||` `^`
- Arithmetic: `*` `/` `+` `-` `%` `<<` `>>`
- Relational: `==` `!=` `<` `>` `>=` `<=`

## ■ Not exactly C:

- For bool: `&` and `|` are logical with full evaluation
- For integer: `&` and `|` perform binary AND/OR

## ■ Very un-C:

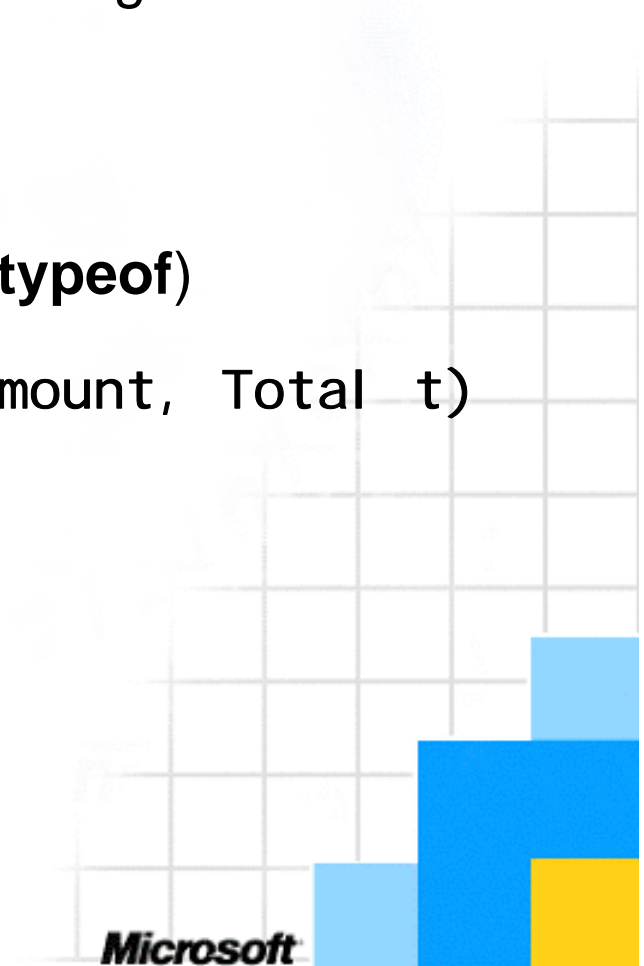
- `is` Tests run-time type
- `as` Converts a value to a specified type
- `typeof` Retrieves run-time type



# Operator Overloading

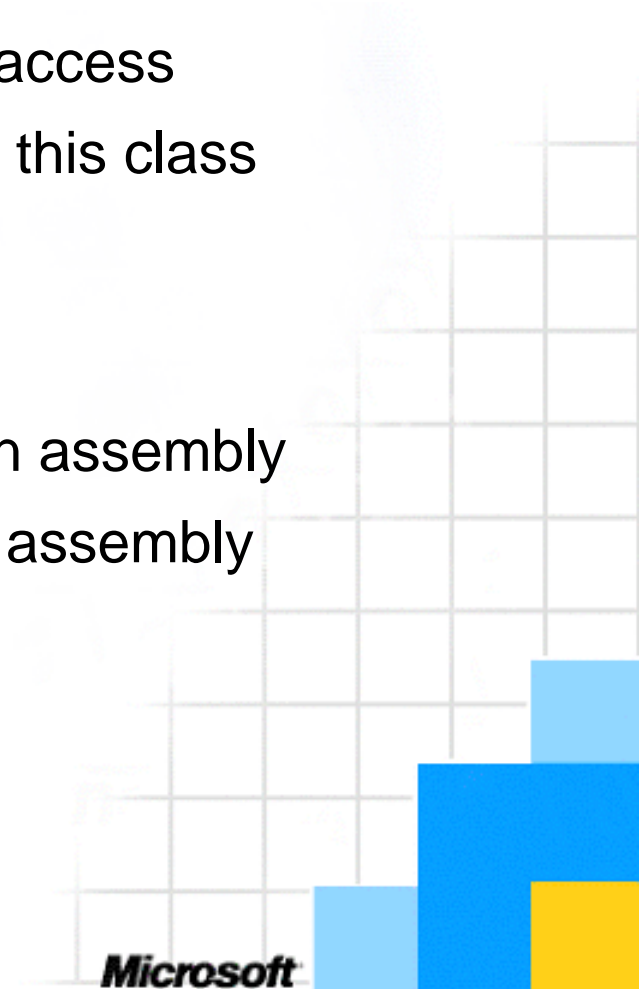
- Most operators can be overloaded
  - Arithmetic, relational, conditional, and logical
- No overloading for
  - Assignment operators
  - Special operators (**sizeof**, **new**, **is**, **typeof**)
- Example: Total operator `+(int Amount, Total t)`

```
{  
    t.total += Amount;  
}
```



# Access Protection

- Adopts C++ model
  - **public** ⇒ Everyone may call or access
  - **protected** ⇒ Only members may access
  - **private** ⇒ Only members of exactly this class
- Expands C++ model
  - **sealed** ⇒ Can't use as base class
  - **internal** ⇒ Public access only within assembly
  - **protected internal** ⇒ Protected in assembly



# ■ “Pointers, I need pointers!”

## ■ C# supports

- Intrinsic **string** type

- Rich garbage-collection model

- By-reference parameters using **ref**

```
void increment(ref int value, int by)
```

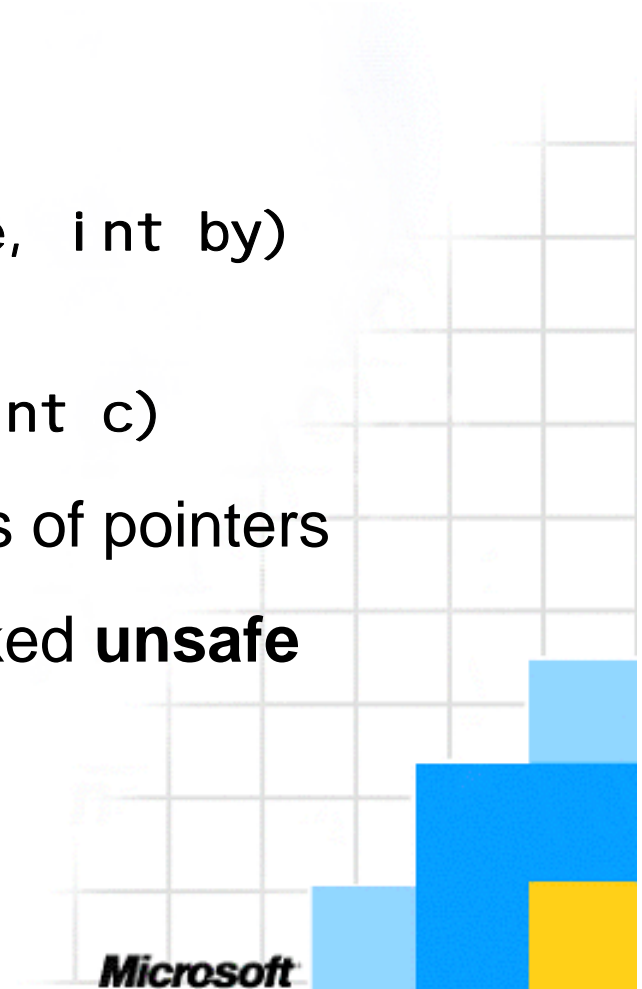
- Outbound parameters using **out**

```
bool add(int a, int b, out int c)
```

## ■ Eliminates vast majority of C++ uses of pointers

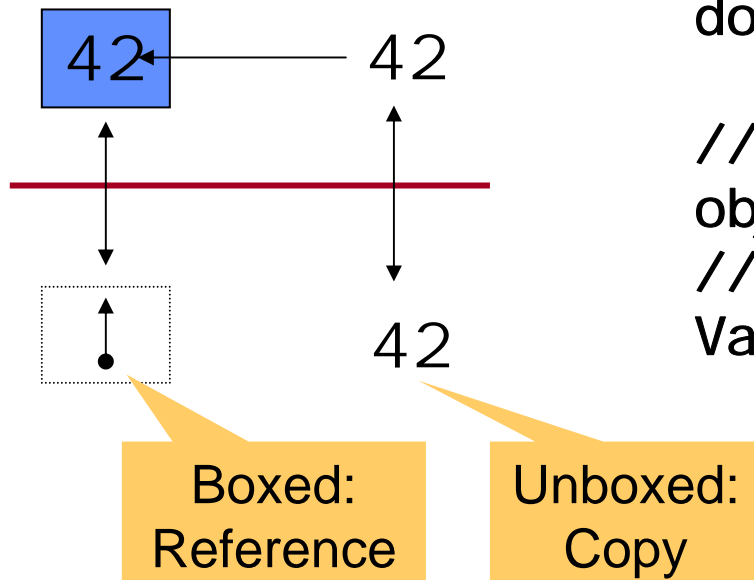
## ■ Pointers *are* available for code marked **unsafe**

```
unsafe void crypt(byte[] arr)
{
    byte * b = arr;
    ...
}
```



# Boxing and Unboxing

- By-value types can be "boxed" and "unboxed"
- "Boxing" allows by-value types to travel by-ref
- Based on objectness of all types.
- Think: Throw value in box and reference the box



```
double Value;  
  
// Boxing  
object BoxedValue = Value;  
// Unboxing  
Value = (double)BoxedValue;
```

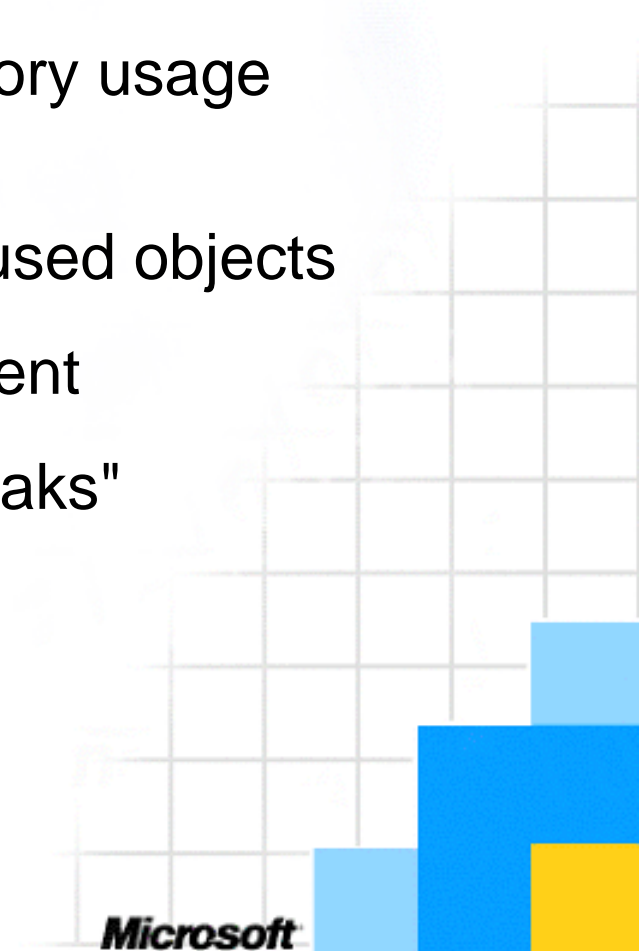
# ■ Garbage Collection 1/2

- C/C++ applications often leak memory
  - Manual memory management
  - No clear rules for ownership of objects
- COM fixes this partly through ref-counts
  - AddRef/Release calls must be balanced
  - SmartPointers don't catch all problems
- Server applications must be leak free
  - Must run for months and years
- Solution: automatic memory management



## ■ Garbage Collection 2/2

- Developer creates new objects and data arrays
  - Everything created/allocated using new keyword
- The .NET runtime tracks all memory usage automatically
- GC automatically removes all unused objects
- More efficient memory management
- Ease of use and "zero memory leaks"



# ■ Nondeterministic Destruction

- Garbage Collection downside:
  - Destructors called at some random future time
  - Finalization code is never synchronous
  - Breaks some established design patterns from C++
- Beware: Code should never depend on destructors to free external resources.

- Instead: Implement `IDisposable` with `using`

```
using( File f = new File("c:\\xyz.xml") )  
{  
    ...  
}
```

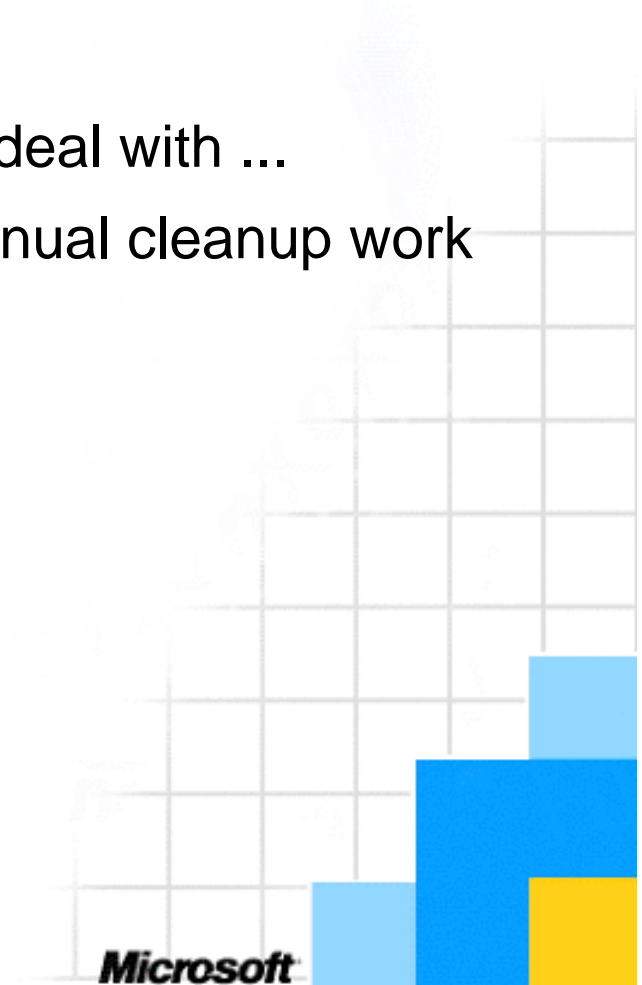
- `IDisposable.Dispose()` called on object when scope is exited.

# ■ Exception Handling

- Very similar to C++ and SEH
- Read like this:
  - **try** running this code ...
  - ... if error occurs, **catch** what I can deal with ...
  - ...**finally** allow me to do some manual cleanup work

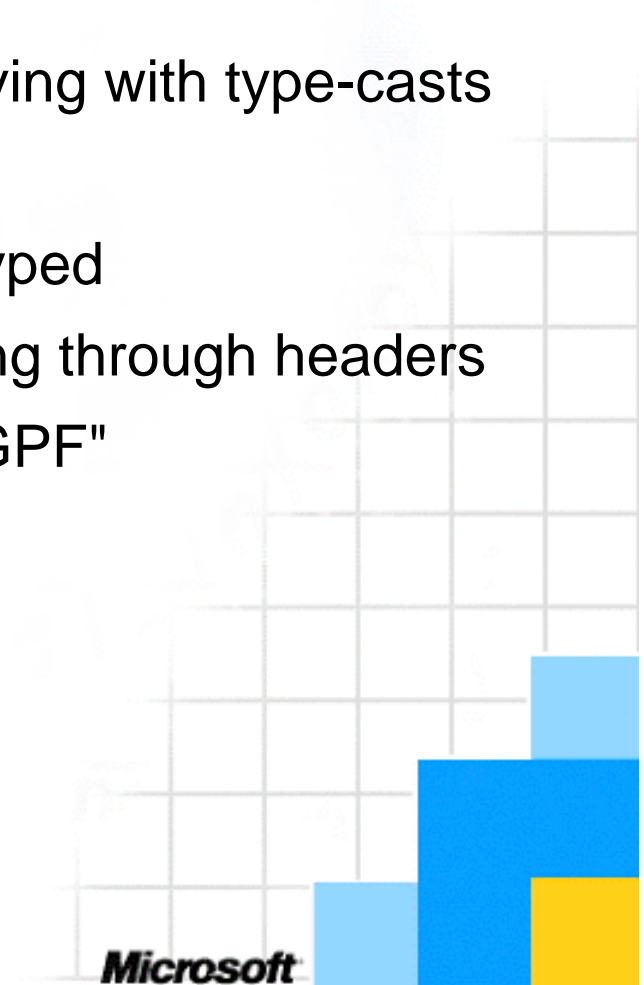
■ Example:

```
try
{
    //... run code
}
catch(SomeException e)
{
    //... handle
}
finally
{
    //...end gracefully
}
```



# ■ Core Differences from C++

- C# looks a lot like C/C++
- Fixes a lot of common bug sources:
  - Stricter type-checking, very unforgiving with type-casts
  - No "fall-through" in **switch**
  - Boolean expressions are strongly typed
  - Better access protection, no cheating through headers
  - "Mostly no pointers" ⇒ "Mostly no GPF"
  - Forget hunting memory leaks



# Class Version Management

- Real Life:
  - Two people at two places write two pieces of software
  - A's class builds on B's class
  - B implements a method **CalcResult**
  - Next version, A also adds **CalcResult**
    - **Q:** Does B want **CalcResult** to be an override of A's ?
    - **A:** Unlikely.
- Solution: Must state intent when using inheritance
  - Must specify **override** to override a method
  - Can specify **new virtual** to never inherit from base



# Goodies: XML Comments

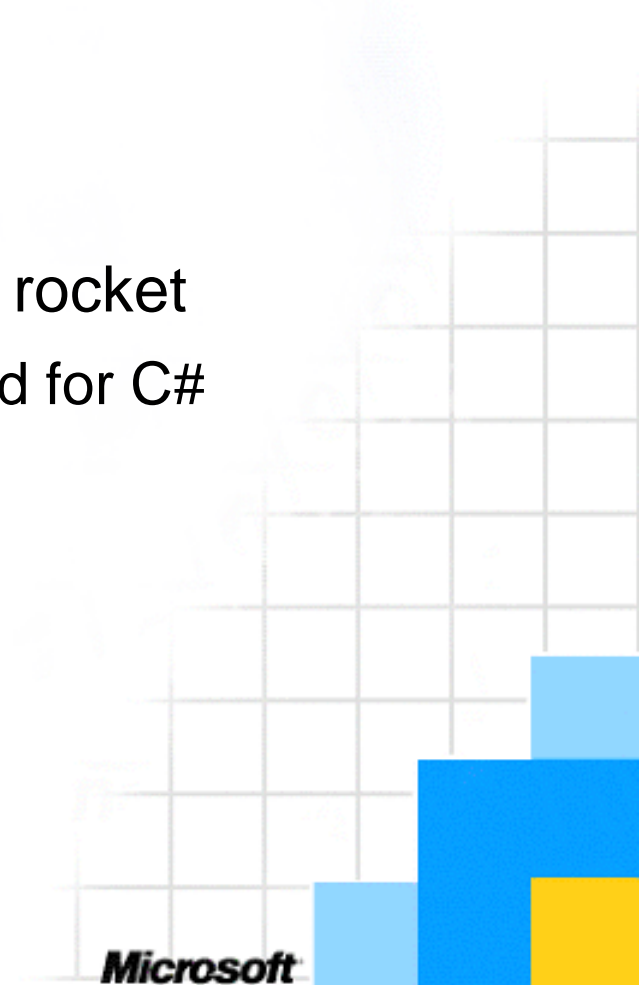
- Consistent, intrinsic way to create doc from code
- Triple-slash "///" comments are exported
- Extract full documentation during compile with **/doc**
- Comes with predefined schema

- Example: 

```
///<summary>
/// This function serves to calculate the
/// overall value of the item including all
/// taxes
/// </summary>
/// <remarks>
/// Tax calculates in CalcTax()
/// </remarks>
public decimal GetTotalValue()
{
}
```

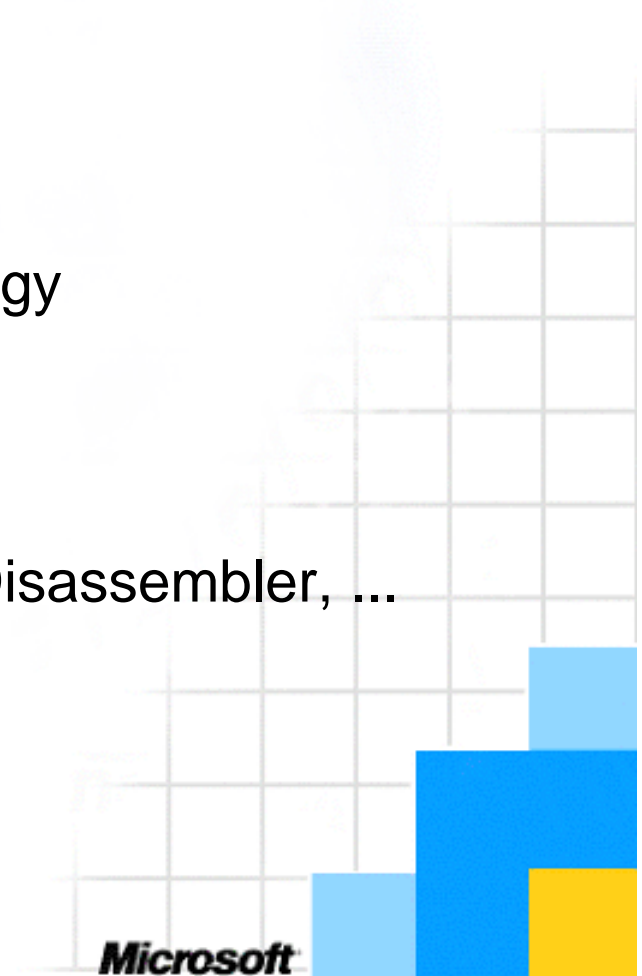
## Section 3: The Tools

- .NET Framework SDK—all you need to build apps
  - C# compiler
  - Visual debugger
  - Nmake
- Visual Studio.NET—the productivity rocket
  - Development environment optimized for C#
  - Code wizards and templates
  - Smart help



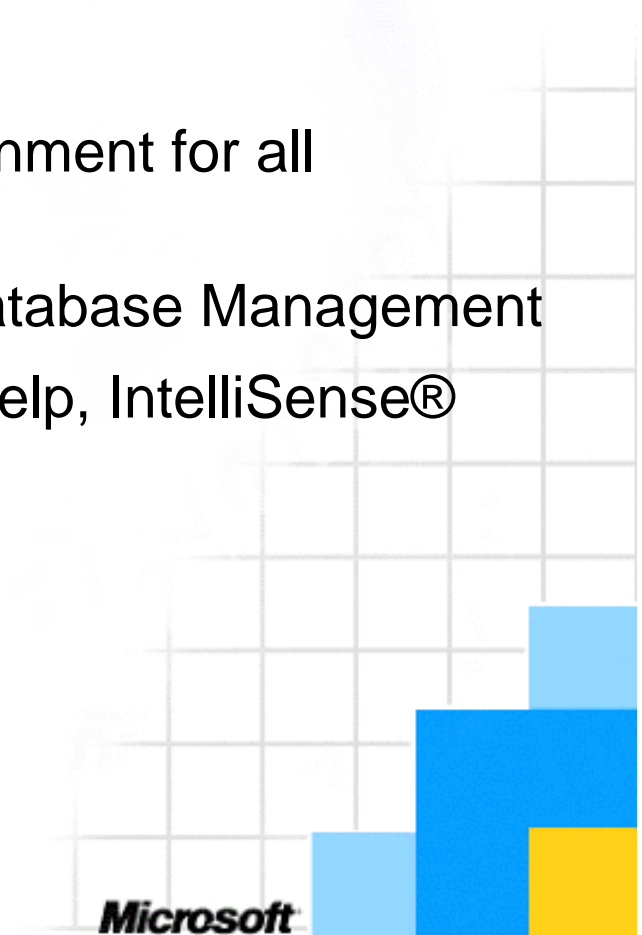
# .NET Framework SDK

- C# compiler (plus Visual Basic, C++, and JScript)
  - All language features
  - Runs from the command line
- Visual Debugger—GuiDebug
  - Built on Visual Studio.NET technology
  - Full access to runtime metadata
- Tools
  - Nmake, security, configuration, IL Disassembler, ...
- Free for everyone



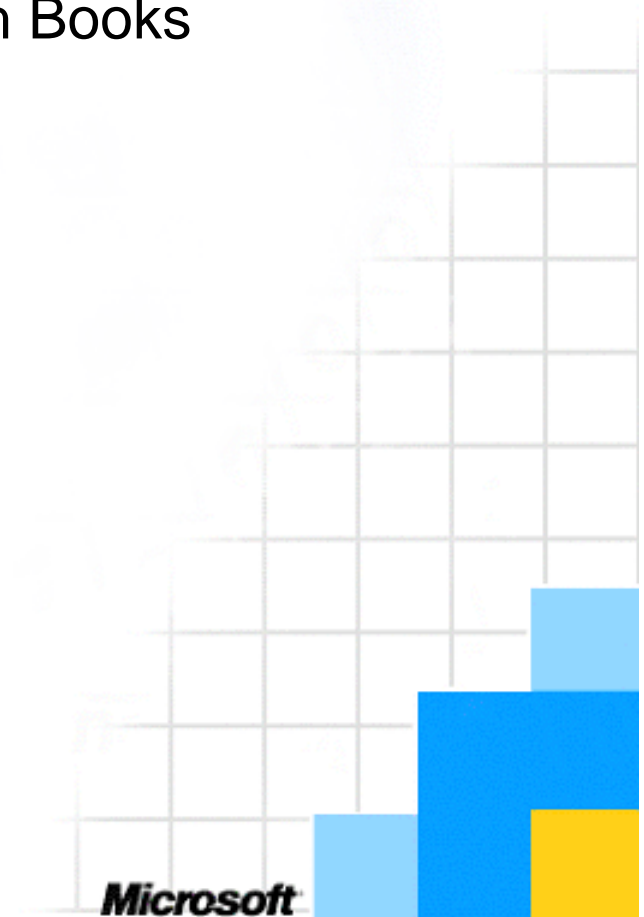
# Visual Studio.NET

- Built on the .NET Framework SDK
- Reinvention of the concept of Visual Studio®, now with:
  - Multiple-language projects
  - One integrated development environment for all languages and tasks
  - Integrated tools: Visual Modeler, Database Management
  - Perfect help integration: Dynamic Help, IntelliSense®
- Highest productivity for all:
  - Rapid application development
  - Large-scale projects



# Section 4: Putting It All Together

- Samples
  - The Classic Start: "Hello World!" in C#
  - Exploring C# Features in Duwamish Books

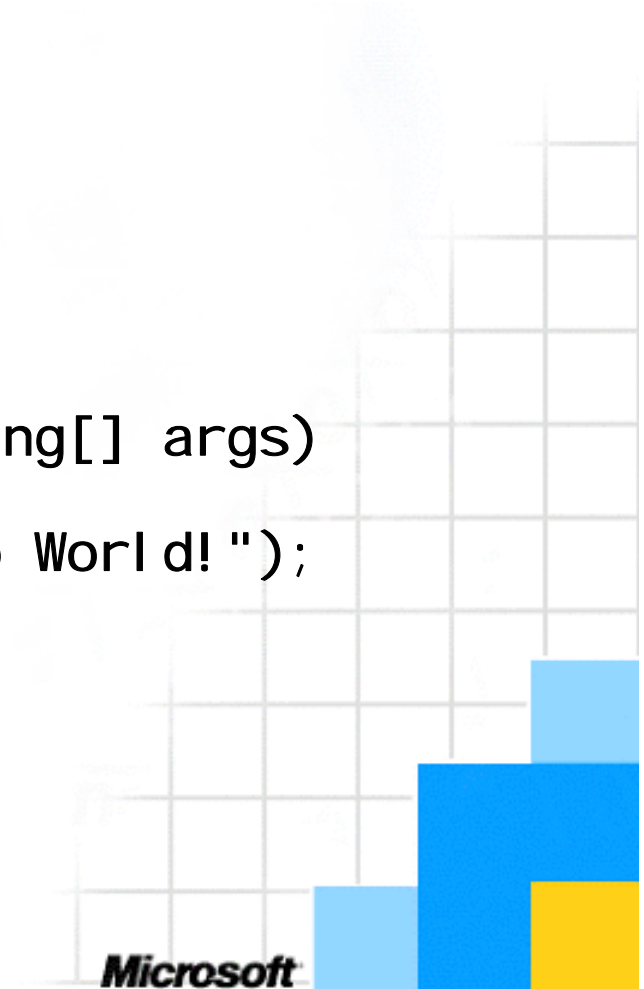


# Hello World

```
namespace Sample
{
    using System;

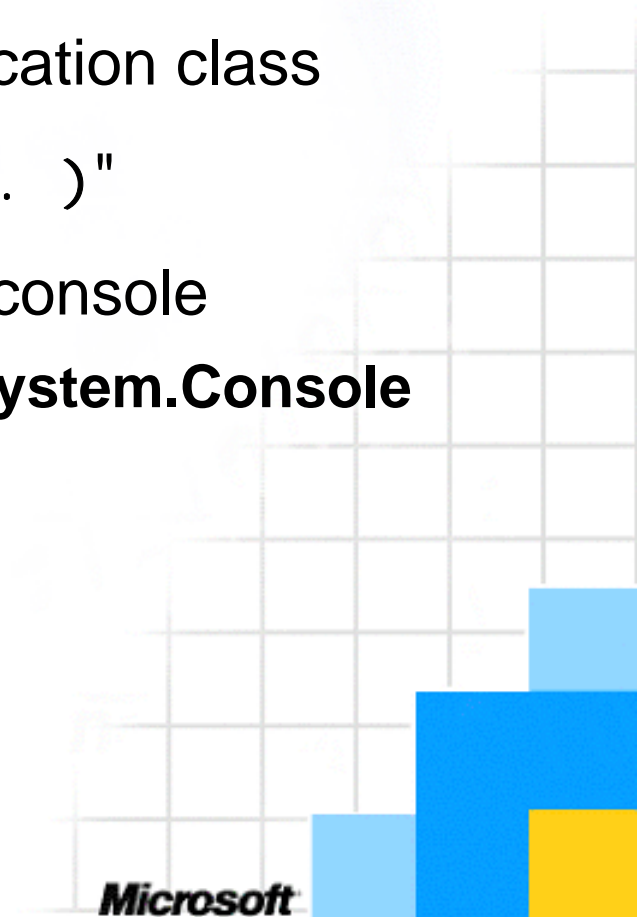
    public class HelloWorld
    {
        public HelloWorld()
        {
        }

        public static int Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            return 0;
        }
    }
}
```



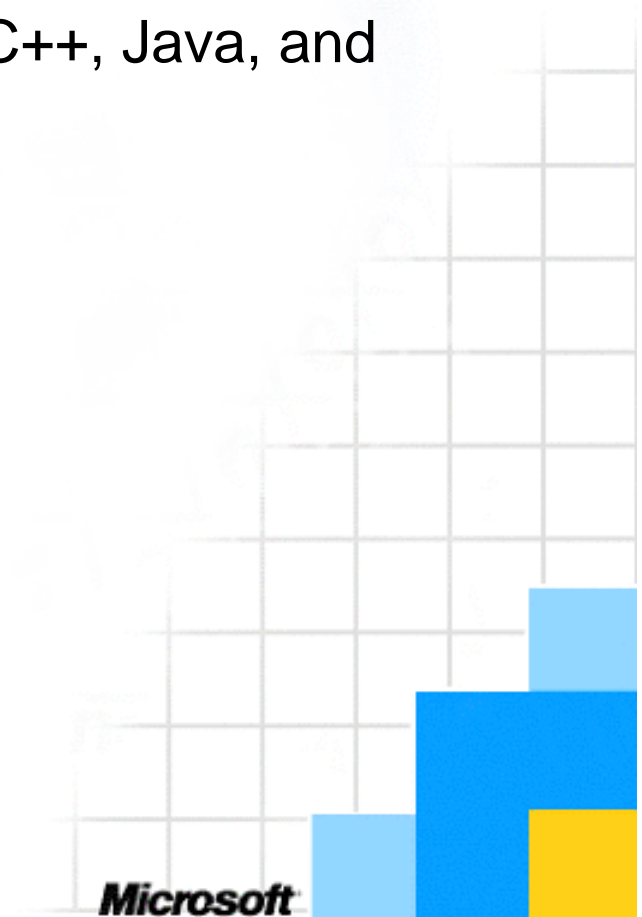
# ■ Hello World Anatomy

- Contained in its own namespace
- References other namespaces with "using"
- Declares a publicly accessible application class
- Entry point is "static int Main( ... )"
- Writes "Hello World!" to the system console
  - Uses static method **WriteLine** on **System.Console**

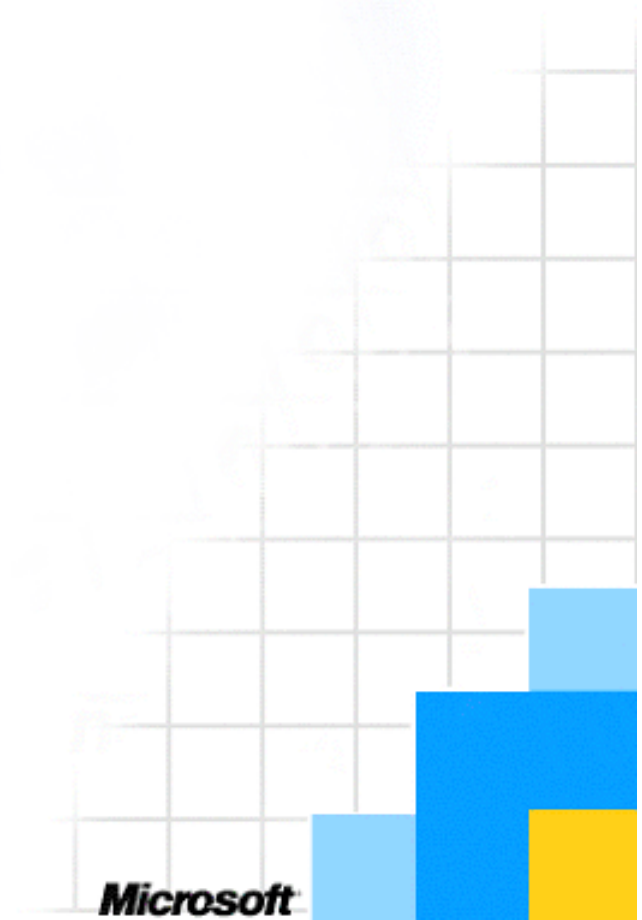


# Summary

- C# builds on the .NET Framework component model
- New language with familiar structure
  - Easy to adopt for developers of C, C++, Java, and Visual Basic applications
- Fully object oriented
- Optimized for the .NET Framework



# Questions?



# To C# from Visual Basic

A very brief introduction to the C language family syntax for Visual Basic developers

# C

- Core principles: Be brief. Be expressive.
  - Relatively few and short keywords
  - Uses symbol ASCII characters instead of words
- Core element: "{" The Block "}"
  - Code groups for structural or flow-control elements

## Blocks in Visual Basic

```
Sub Xyz()  
    ' Code  
End Sub
```

```
If MyVal = 0 Then  
    ' Code  
End If
```

## Blocks in C

```
void Xyz() {  
    /*Code*/  
}
```

```
if (MyVal == 0) {  
    /*Code*/  
}
```

# Statements

- All statements ("do that!") are inside blocks
  - Every statement is terminated with a semicolon ";"
  - White space, line breaks, tabs are not significant
  - May have many statements on one line
  - Statements may be split across lines

Statements:

```
a = add( 2, 3 );  
q = a - 1;  
out( q );  
q--;  
return;
```

Just as legal:

```
a=  
add  
(2, 3  
); q=a  
-1; out  
(q); q--  
; return;
```

This too:

```
main() {  
    int i=0;  
    while(i<10) {  
        printf("i=%d\n", i);  
        i++;  
    }  
}
```

# Declaration Principles

- Core declaration principle: "What then who"
- What: data type or declaration keyword
  - Types: **int**, **string**, **decimal**, **float**
  - Keywords: **enum**, **struct**, **class**
- Who: user-defined identifier
  - Must begin with a letter, may contain numbers, "\_"
  - **Attention:** All identifiers are case sensitive

Variable declarations:

```
int MyValue;  
string MyName;
```

Function declaration:

```
int MyFunc(string Arg);
```

Structure declaration:

```
struct Point {  
    int x; int y;  
}
```

# Operators

## ■ Arithmetic

- binary operators → "opr1 operator opr2"
  - add (+), subtract (-), divide (/), multiply (\*)
  - Modulo (%) →  $a=b\%c$ ; instead of  $a=b \text{ mod } c$
- unary operators → "opr1 operator"
  - increment (++), decrement (--) →  $i++$  instead of  $i=i+1$

## ■ Logical

- and (&&), or (||), not(!)

## ■ Bitwise

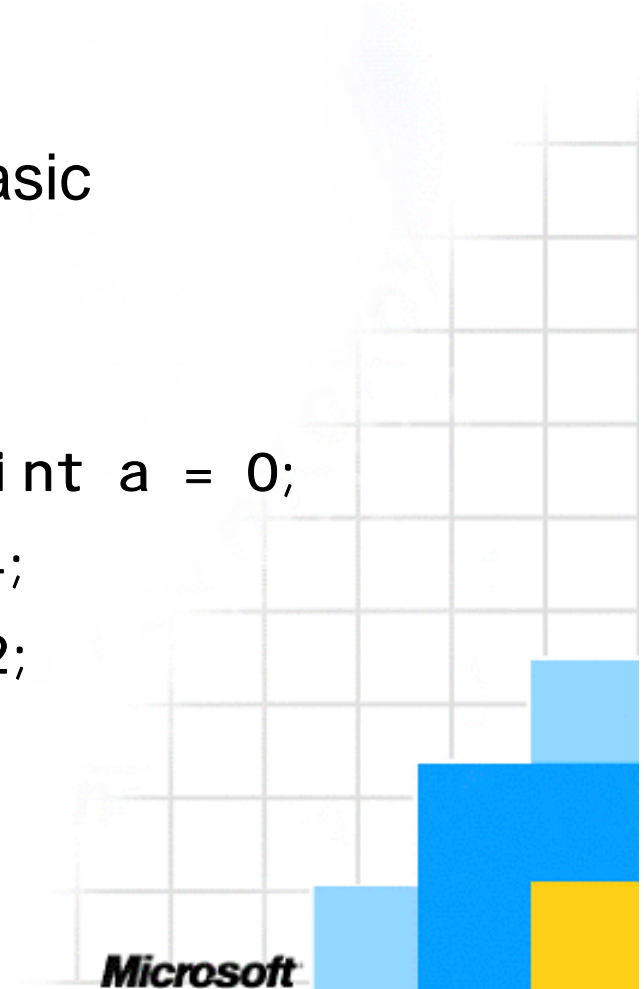
- and (&), or(|), xor (^), not (~)

## ■ Relational

- equal (==), not equal (!=), less (<)
- greater (>), less equal (<=), greater equal (>=)

# Expressions

- Boolean expressions
  - Evaluate to **true** or **false**
  - $(2 == 1)$  is **false**,  $(2 > 1)$  is **true**
- Arithmetic expressions like Visual Basic
  - $1 + 2 * (6 / 2)$
- Assignment
  - $a = 1;$   $\Rightarrow$  also at declaration time: `int a = 0;`
  - $a += 4;$   $\Rightarrow$  equivalent to  $a = a + 4;$
  - $a *= 2;$   $\Rightarrow$  equivalent to  $a = a * 2;$ 
    - works with all binary operators.



# Flow Control

- Conditional execution of code blocks:
  - `if (<expression>) <block> [else <block>];`
- Selective execution of code blocks:
  - `switch(<variable>) {  
    case <value>:  
        <statements>; break;  
}`

```
if (i == 0 )  
{  
    Console.WriteLine("This");  
}  
else  
{  
    Console.WriteLine("That");  
};
```

```
switch (i)  
{  
    case 0:  
        Console.WriteLine("This");  
        break;  
    case 1:  
        Console.WriteLine("That");  
        break;  
    default:  
        Console.WriteLine("Else");  
        break;  
}
```

# Loops

## ■ Counted loops

- `for(<pre>; <while>; <increment>) <block>`
- `<pre>`: Setting precondition "`i =0`"
- `<while>`: Testing continuation condition "`i <10`"
- `<increment>`: Calculating counter state "`i ++`"

## ■ While loops

- `while (<expression>) <block>`
- `do <block> while (<expression>);`

```
for (i =0; i < 10; i ++ )  
{  
    Console.Wri te(i);  
}
```

```
while ( !MyFi le. EOF )  
{  
    Console.Wri te(MyFi le. Read());  
}
```

## Other Noteworthy Things ...

- The data type "void" means "no type"
  - $\Rightarrow$  method `void func(int a)` returns nothing
    - Like a "Sub" in Visual Basic
  - $\Rightarrow$  method `int func(void)` has no arguments
- C has pointers. In C# you do not need them
  - $\Rightarrow$  Don't worry about them
  - Reference and value types are clearly defined
  - **ref** keyword for values by reference
- Very few keywords and intrinsic functionality
  - "C" languages rely heavily on run-time libraries
  - Example: `Math.Pow(a, b)` instead of  $a^b$

# Legal Notices

Unpublished work. © 2001 Microsoft Corporation. All rights reserved.

Microsoft, IntelliSense, JScript, Visual Basic, Visual Studio, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

