

# .Net Training

Basics of .Net (C#)



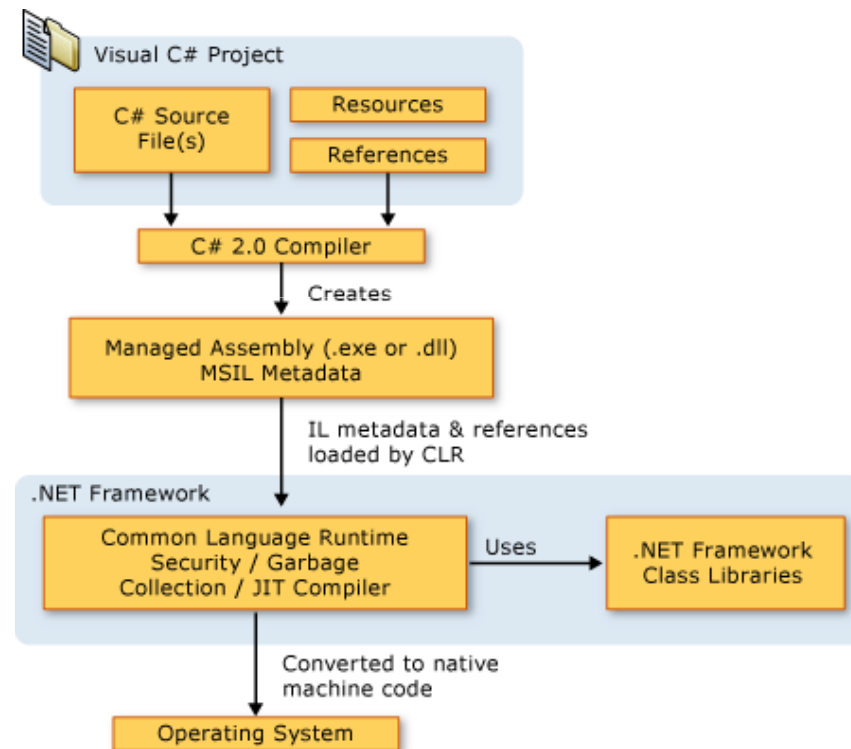
# Agenda

---

- .Net Overview
- Anatomy of a C# program
- C# Syntax
- Assemblies
- Global Assembly Cache
- Namespaces
- Data Types
- Classes and Structs
- Enumerators
- Access Modifiers
- Properties
- Scoping
- Operators
- .Net Framework Design Guidelines
- Resources

# .Net Overview

- .Net is a development and runtime platform
- A “Virtual Machine” interacts between the O/S and your programs
- Your programs depend on the “Virtual Machine” – the .Net Framework
- .Net Framework provides services in Security, Garbage Collection, JIT Compiler, Class Libraries and other infrastructure
- After it is run for the first time the program is JIT compiled into native code – that makes it fast



# .Net Overview

---

- Does that mean .Net code is as fast as assembly code or C++ code?
  - **NO!**
  - You wouldn't want to write an RDBMS with it
  - But it is much faster than interpreted code like Java
  - And for most business applications it is fast enough

# .Net Overview

---

- C# syntax is highly expressive, yet with less than 90 keywords, it is also simple and easy to learn.
- The curly-brace syntax of C# will be familiar to C, C++, Java or JavaScript Developers.
- Developers who know any of these languages are typically able to begin working productively in C# within a very short time.
- C# syntax simplifies many of the complexities of C++ while providing powerful features such as nullable value types, enumerations, delegates, anonymous methods and direct memory access, which are not found in Java.
- As an object-oriented language, C# supports the concepts of encapsulation, inheritance and polymorphism.
- All variables and methods, including the Main method, the application's entry point, are encapsulated within class definitions.
- A class may inherit directly from one parent class, but it may implement any number of interfaces.

# .Net Overview

---

- Another key feature of .Net is the lack of use of the Registry.
- Components (DLL's) are stored in the same folder as the executing program (EXE)
- You can have multiple versions of the same Library (DLL) in different folders on the same machine side-by-side
- If you need to share Libraries you can store them in the Global Assembly Cache
- This avoids DLL Hell!
- Enables XCOPY deployment



# Anatomy of a C# Program

- “using” statements at top
- Namespace declaration
- Class declaration
- Field declaration (variables)
- Constructor
  - (comments marked with //)
- Optional Destructor
- Property
- Method

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

namespace Day02_BasicsOfDotNet.Exercises
{
    class HelloWorld
    {
        string text;

        HelloWorld()
        { //constructor
            text = "Hello World!";
        }

        ~HelloWorld()
        { //destructor
            text=null;
        }

        string Text
        {
            get {return text;}
            set { text = value; }
        }

        string Run()
        {
            return text;
        }
    }
}
```

# C# Syntax

---

- Curly braces delineate start and end of a code block ({})
- Semi-colon ends every statement (;)
- A statement is a procedural building-block from which all C# programs are constructed.
  - A statement can declare a local variable or constant, call a method, create an object, or assign a value to a variable, property, or field. A control statement can create a loop, such as a for loop, or make a decision and branch to a new block of code, such as an if or switch statement. Statements are usually terminated by a semicolon. For more information, see Statement Types (C# Reference).
- A series of statements surrounded by curly braces form a block of code.
  - A method body is one example of a code block. Code blocks often follow a control statement. Variables or constants declared within a code block are only available to statements within the same code block.

- In the Method:

```
string Run()  
{  
    return text;  
}
```

- This is a Code Block:

```
string Run()  
{  
    ...  
}
```

- This is a Statement:

```
return text;
```

# C# Syntax

- C# programs can consist of one or more files.
- Each file can contain zero or more namespaces.
- A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

# Assemblies

---

- An assembly is a fundamental building block of any .NET Framework application.
- For example, when you build a simple C# application, Visual Studio creates an assembly in the form of a single portable executable (PE) file, specifically an EXE or DLL.
- Assemblies contain metadata that describe their own internal version number and details of all the data and object types they contain.
- Assemblies are only loaded as they are required. If they are not used, they are not loaded.
- This means that assemblies can be an efficient way to manage resources in larger projects.
- Assemblies can contain one or more modules.

# Assemblies

---

- Assemblies are implemented as .exe or .dll files.
- You can share an assembly between applications by placing it in the Global Assembly Cache.
- Assemblies are only loaded into memory if they are required.
- You can programmatically obtain information about an assembly using Reflection.
- You can use two versions of the same assembly in a single application.

# Global Assembly Cache

---

- Each computer where the common language runtime is installed has a machine-wide code cache called the Global Assembly Cache. The GAC stores assemblies specifically designated to be shared by several applications on the computer.
- You should share assemblies by installing them into the GAC only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required.

# Namespaces

---

- Namespaces are heavily used in C# programming in two ways:
- First, the .NET Framework uses namespaces to organize its many classes, as follows:
  - `System.Console.WriteLine("Hello World!");`
- **System** is a namespace and **Console** is a class contained within that namespace.
- The **using** keyword can be used so that the entire name is not required, like this:
  - `using System;`
  - `Console.WriteLine("Hello World!");`

# Namespaces

---

- Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace MyNameSpace
{
    class MyClass
    { ... }
}
```

# Namespaces

---

- A namespace has the following properties:
  - They organize large code projects.
  - They are delimited with the . operator.
  - The using directive means you do not need to specify the name of the namespace for every class.
  - The global namespace is the "root" namespace: `global::system` will always refer to the .NET Framework namespace `System`.

# Data Types

---

- A data type can be described as being either:
  - A built-in data type, such as an int or char, or
  - A user-defined data type, such as a class or interface.
  - Data types can also be defined as being either:
    - Value Types (C# Reference), which store values, or
    - Reference Types (C# Reference), which store references to the actual data.

# Data Types (Value Types)

---

- The value types consist of two main categories:
  - Structs
  - Enumerations
- Structs fall into these categories:
  - Numeric types
    - Integral types
    - Floating-point types
    - decimal
  - bool
  - User defined structs.

# Data Types (Value Types)

---

- Variables that are based on value types directly contain a value.
  - Assigning one value type variable to another copies the contained value.
  - This differs from the assignment of reference type variables, which copies a reference to the object but not the object itself.
- All value types are derived implicitly from the `System.ValueType`.
- Unlike reference types, it is not possible to derive a new type from a value type. However, like reference types, structs can implement interfaces.
- Unlike reference types, it is not possible for a value type to contain the null value. However, the nullable types feature does allow values types to be assigned to null.
- Each value type has an implicit default constructor that initializes the default value of that type.
- Simple types can be initialized using literals. For example, 'A' is a literal of the type **char** and 2001 is a literal of the type **int**.

# Data Types (Simple Types)

---

- All of the simple types - those integral to the C# language - are aliases of the .NET Framework System types
- The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespace.

# Data Types (Simple Types)

C# Type	.NET Framework Type	Approx. Range	Precision	Size
bool	System.Boolean	true or false	N/A	1 bit
byte	System.Byte	0 to 255	N/A	Unsigned 8-bit integer
sbyte	System.SByte	-128 to 127	N/A	Signed 8-bit integer
char	System.Char	U+0000 to U+ffff	N/A	Unicode 16-bit character
decimal	System.Decimal	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$	28-29 significant digits	128-bit number
double	System.Double	$\pm 5.0 \times 10-324$ to $\pm 1.7 \times 10308$	15-16 digits	64-bit number
float	System.Single	$\pm 1.5 \times 10-45$ to $\pm 3.4 \times 1038$	7 digits	32-bit number
int	System.Int32	-2,147,483,648 to 2,147,483,647	N/A	Signed 32-bit integer
uint	System.UInt32	0 to 4,294,967,295	N/A	Unsigned 32-bit integer
long	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	N/A	Signed 64-bit integer
ulong	System.UInt64	0 to 18,446,744,073,709,551,615	N/A	Unsigned 64-bit integer
short	System.Int16	-32,768 to 32,767	N/A	Signed 16-bit integer
ushort	System.UInt16	0 to 65,535	N/A	Unsigned 16-bit integer

# Data Types (Literals)

---

- Literals can be used for assigning values to simple type variables:
  - `int i = 10;`
  - `char char1 = 'Z'; // Character literal`
  - `char char2 = '\x0058'; // Hexadecimal`
  - `char char3 = (char)88; // Cast from integral type`
  - `char char4 = '\u0058'; // Unicode`
  - `decimal d = 10 // decimal literal`
  - `decimal d = 10.5m //Without the suffix m, the number is treated as a double, thus generating a compiler error`
  - `decimal d = (decimal)10.5 //Cast from double type`

# Data Types (Reference Types)

---

- Variables of reference types, referred to as objects, store references to the actual data.
- These are the keywords used to declare reference types:
  - class
  - interface
  - delegate
- There are two main built-in reference types:
  - object
  - string

# Data Types (Reference Types)

---

- The main difference between Value and Reference Types in terms of how they work in the computer is:
  - Value Types contain values that are stored in the Stack
  - Reference Types contain a reference to a location in memory in the Stack. The actual value is stored in the Heap.
  - This gives a speed advantage to .Net when dealing with Value Types

# Data Types (Reference Types)

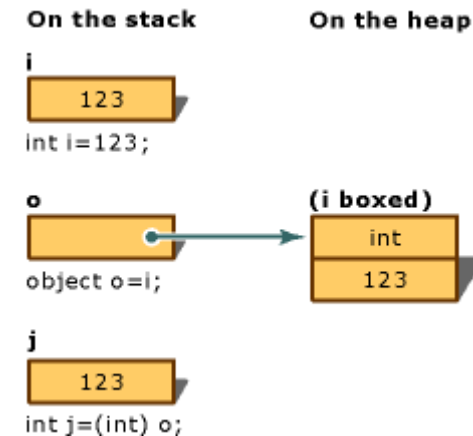
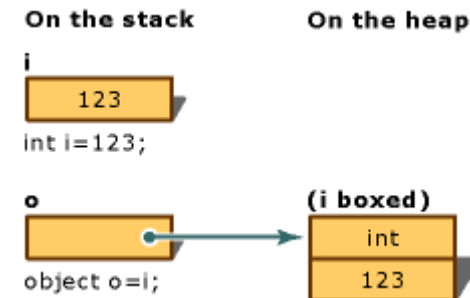
---

- So how do Value Types and Reference Types interact with each other?
  - Boxing
  - Unboxing
- Boxing a value type packages it inside an instance of the Object reference type. This allows the value type to be stored on the garbage collected heap.
- Unboxing extracts the value type from the object.

# Data Types (Reference Types)

```
int i = 123;  
object o = (object) i; // boxing
```

```
o = 123;  
i = (int) o; // unboxing
```



# Data Types (Casting)

---

- You can cast a data type to another data type in most cases.
  - decimal a = 100.45m; //100.45
  - double b = (double)a; //100.45
  - int c = (int)b; //100
  - decimal d = (decimal)c; //100
- Some conversions are “lossy”
- Some are impossible
  - Cat charlie = new Cat();
  - Employee emp = (Employee)charlie; //ERROR!

# Data Types as Parameters

- Passing parameters to methods follows certain rules
- By default parameters are passed in “by value”
- But you can also pass parameters by reference using the **ref** and **out** keywords
- When you pass by value the data is not changed
- When you pass by reference the data is changed

```
public class Test
{
    void Main()
    {
        int a = 10;
        int b = 20;
        int c;
        RunTest(a, ref b, out c);
        Console.WriteLine("a: {0}\r\nb: {1}\r\nc: {2}", a, b,
c);
        //a: 10
        //b: 40
        //c: 60
    }

    void RunTest(int a, ref int b, out int c)
    {
        a += 10;
        if (a > 0)
        {
            b *= 2;
        }
        c = b + a;
    }
}
```

# Data Types as Parameters

---

- For Value Types:
  - Passing by value makes a copy of the value of the field to pass into the parameter
  - Passing by reference passes the address where the value is located, allowing the value to be changed
  - Passing by out is like passing by ref, but the method is responsible for assigning the out parameter before control is returned
- For Reference Types
  - Passing by value passes a copy of the pointer to the memory location; changing the data in the object affects the caller's reference as well
  - Passing by reference passes the original pointer. Effects same as above, but subtly different
  - Passing by out is the same as by ref, but the method must assign the parameter before control is returned

# Passing Val Type by val

---

```
int a = 10;  
Run(a);  
//a=10;
```

## Stack

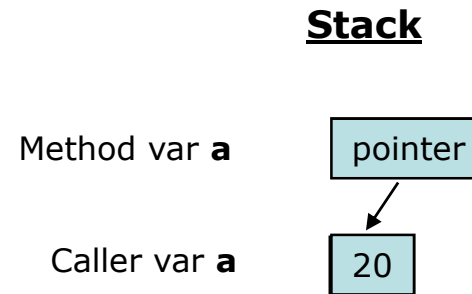
Method var <b>a</b>	20
Caller var <b>a</b>	10

```
void Run(int a)  
{  
    //a=10;  
    a=20;  
    //a=20  
}
```

# Passing Val Type by ref

```
int a = 10;  
Run(ref a);  
//a=20;
```

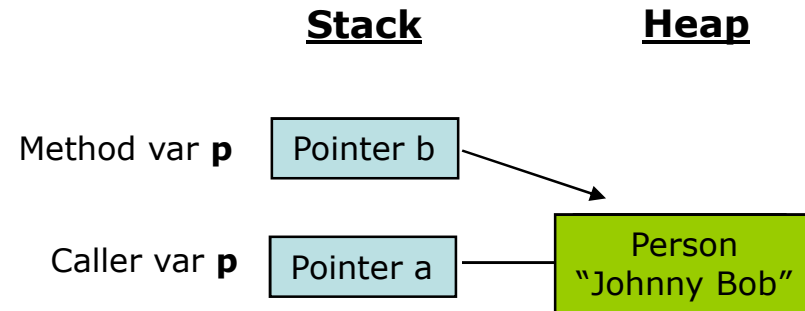
```
void Run(ref int a)  
{  
    //a=10;  
    a=20;  
    //a=20  
}
```



# Passing Ref Type by val

```
Person p = new Person();  
p.FullName="Billy Bob";  
Run(p);  
//p.FullName="Johnny Bob";
```

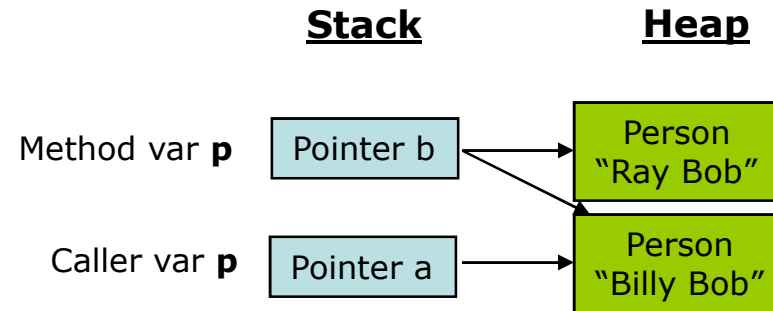
```
void Run(Person p)  
{  
    p.FullName="Johnny Bob"  
}
```



# Passing Ref Type by val

```
Person p = new Person();  
p.FullName="Billy Bob";  
Run(p);  
//p.FullName="Billy Bob";
```

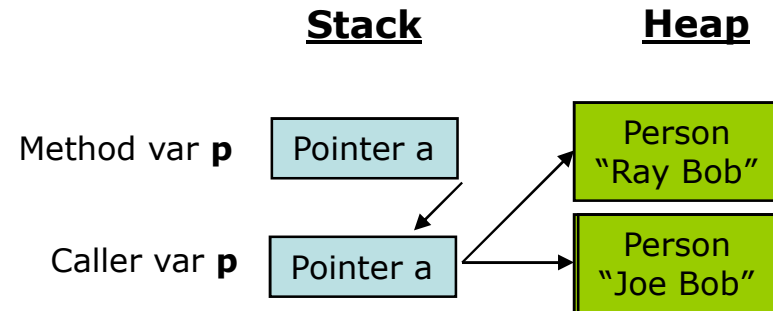
```
void Run(Person p)  
{  
    p = new Person();  
    p = "Ray Bob";  
}
```



# Passing Ref Type by ref

```
Person p = new Person();  
p.FullName="Billy Bob";  
Run(ref p);  
//p.FullName="Ray Bob";
```

```
void Run(ref Person p)  
{  
    p.FullName="Joe Bob";  
    p = new Person();  
    p = "Ray Bob";  
}
```



# Classes and Structs

---

- The main object types you create will be Classes
- C# is an object-oriented programming language, and uses classes and structs to implement types such as Windows Forms, user interface controls, and data structures.
- A typical C# application consists of classes defined by the programmer, combined with classes from the .NET Framework.
- C# provides many powerful ways of defining classes, such as providing different access levels, inheriting features from other classes, and allowing the programmer to specify what happens when types are instantiated or destroyed.

# Classes and Structs

---

- Objects, classes and structs have the following properties:
  - Objects are instances of a given data type. The data type provides a blueprint for the object that is created — or instantiated — when the application is executed.
  - New data types are defined using classes and structs.
  - Classes and structs form the building blocks of C# applications, containing code and data. A C# application will always contain at least one class.
  - A struct can be considered a lightweight class, ideal for creating data types that store small amounts of data, and does not represent a type that might later be extended via inheritance.
  - C# classes support inheritance, meaning they can derive from a previously defined class.

# Classes and Structs

---

- Classes are defined using the class keyword, as shown in the following example:

```
public class Customer
```

```
{
```

```
    //Fields, properties, methods and events go here...
```

```
}
```

- Classes defined Reference Type objects

# Classes and Structs

---

- Structs are defined using the struct keyword, as shown in the following example:

```
public struct PostalAddress
{
    // Fields, properties, methods and events go here...
}
```

- Structs define Value Type objects

# Enumerations (Enums)

- The **enum** keyword is used to declare an enumeration
- It is a distinct type consisting of a set of named constants called the enumerator list.
- Every enumeration type has an underlying type, which can be any integral type except char.
- The default underlying type of the enumeration elements is int.
- By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.  
For example:
  - **enum** Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri}; //Sat=0
  - **enum** Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri}; //Sat=1
  - **enum** Days {Sat=7, Sun=1, Mon=2, Tue=3, Wed=4, Thu=5, Fri=6};

# Access Modifiers

---

- The idea behind Encapsulation is to provide a friendly interface to users of your class while hiding the complexity of the internals
- In .Net Access Modifiers help us do that
- Access modifiers are keywords added to the class, struct, or member declaration to specify these restrictions.
- Those keywords are public, private, protected, and internal.

# Access Modifiers

---

- The idea behind Encapsulation is to provide a friendly interface to users of your class while hiding the complexity of the internals
- In .Net Access Modifiers help us do that
- Access modifiers are keywords added to the class, struct, or member declaration to specify these restrictions.
- Those keywords are public, private, protected, and internal.

# Access Modifiers

Declared accessibility	Meaning
<b>public</b>	Access is not restricted.
<b>protected</b>	Access is limited to the containing class or types derived from the containing class.
<b>internal</b>	Access is limited to the current assembly.
<b>protected internal</b>	Access is limited to the current assembly or types derived from the containing class.
<b>private</b>	Access is limited to the containing type.

# Access Modifiers (Example)

- Any code can create an instance of the HelloWorld class.
- Only the HelloWorld class can have direct access to the variable “text”
- Only subclasses of HelloWorld can have access to the property “Text”
- Any code that has an instance of the HelloWorld class can call the Run() method.
- Only code in the same assembly (.exe or .dll) can access the property “MyText”

```
public class HelloWorld
{
    public HelloWorld()
    {}

    private string text;

    protected string Text
    {
        get { return text; }
        set { text = value; }
    }

    public string Run()
    {
        return text;
    }

    internal string MyText
    {
        get { return text; }
    }
}
```

# Properties

---

- In general it is a good idea to keep your variables hidden inside a class and expose them only through Properties
- Properties are members that provide a flexible mechanism to read, write, or compute the values of private fields.
- Properties can be used as though they are public data members, but they are actually special methods called *accessors*.
- This enables data to be accessed easily while still providing the safety and flexibility of methods.



# Properties

---

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}
```

```
class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Assigning the Hours property causes the 'set' accessor to be called.
        t.Hours = 24;

        // Evaluating the Hours property causes the 'get' accessor to be called.
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
```

# Properties

---

- Properties are a concept unique to .Net – Java doesn't have it.
- Inside the Framework, the compiler translates the property:

```
public double Hours
{
    get { return seconds / 3600; }
    set { seconds = value * 3600; }
}
```

- Into two method calls:
  - public double get\_Hours();
  - public void set\_Hours(double value);

# Scoping

---

- Scoping defines visibility for reference of a field (variable) or constant
- Fields (variables) or constants declared within a code block are only available to statements within the same code block
- As soon as the field goes “out-of-scope” it is available for garbage collection.

# Scoping

```
public class Employee
{
    private int yearsOfEmployment = 4; //These fields are available throughout the class
    private decimal salary = 5000;

    public decimal CalcSecondaryBonus()
    { //this field is available only inside this method
        decimal secBonus = yearsOfEmployment * 100;
        return secBonus;
    }

    public decimal CalcMainBonus()
    {
        decimal mainBonus = (salary / 12) * yearsOfEmployment;
        mainBonus += secBonus; //This throws a compiler error
        mainBonus += CalcSecondaryBonus(); //This works
        return mainBonus;
    }
}
```

# Operators

---

- Common operators include:
- == (equality operator) [ if (a==b) ]
- + (addition operator) [ a=b+c]
- - (subtraction operator) [a=b-c]
- \* (multiplication operator) [a=b\*c]
- / (division operator) [a=b/c]
- = (assignment operator) [ int a=10; a=a+b; a+=b; a=a-b; a-=b ; a=a\*b; a\*=b; a=a/b; a/=b]
- Operators can be overloaded

# Operators

Operator category	Operators
Arithmetic	+ - * / %
Logical (Boolean and bitwise)	&   ^ ! ~ &&    true false
String concatenation	+
Increment, decrement	++ --
Shift	<< >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &=  = ^= <<= >>= ??
Member access	.
Indexing	[]
Cast	()
Conditional	?:
Delegate concatenation and removal	+ -
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &

# Design Guidelines

---

- Microsoft posts various design guidelines on MSDN
  - [http://msdn2.microsoft.com/en-us/library/czefa0ke\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/czefa0ke(vs.71).aspx)
- One of the most important ones relates to Naming guidelines
  - [http://msdn2.microsoft.com/en-us/library/czefa0ke\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/czefa0ke(vs.71).aspx)

# Naming Guidelines

---

- [Capitalization Styles](#)
  - Describes the Pascal case, camel case, and uppercase capitalization styles to use to name identifiers in class libraries.
- [Case Sensitivity](#)
  - Describes the case sensitivity guidelines to follow when naming identifiers in class libraries.
- [Abbreviations](#)
  - Describes the guidelines for using abbreviations in type names.
- [Word Choice](#)
  - Lists the keywords to avoid using in type names.
- [Avoiding Type Name Confusion](#)
  - Describes how to avoid using language-specific terminology in order to avoid type name confusion.
- [Namespace Naming Guidelines](#)
  - Describes the guidelines to follow when naming namespaces.
- [Class Naming Guidelines](#)
  - Describes the guidelines to follow when naming classes.
- [Interface Naming Guidelines](#)
  - Describes the guidelines to follow when naming interfaces.

# Naming Guidelines

---

- [Attribute Naming Guidelines](#)
  - Describes the correct way to name an attribute using the Attribute suffix.
- [Enumeration Type Naming Guidelines](#)
  - Describes the guidelines to follow when naming enumerations.
- [Static Field Naming Guidelines](#)
  - Describes the guidelines to follow when naming static fields.
- [Parameter Naming Guidelines](#)
  - Describes the guidelines to follow when naming parameters.
- [Method Naming Guidelines](#)
  - Describes the guidelines to follow when naming methods.
- [Property Naming Guidelines](#)
  - Describes the guidelines to follow when naming properties.
- [Event Naming Guidelines](#)
  - Describes the guidelines to follow when naming events.

# Capitalization Styles

---

- Use the following three conventions for capitalizing identifiers:
- **Pascal case**
  - The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:
    - `BackColor`
- **Camel case**
  - The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:
    - `backColor`
- **Uppercase**
  - All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:
    - `System.IO`
    - `System.Web.UI`

# Capitalization Styles

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException (Note: Always ends with the suffix Exception.)
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable (Note: Always begins with the prefix I.)
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue (Note: Rarely used. A property is preferable to using a protected instance field.)
Public instance field	Pascal	RedValue (Note: Rarely used. A property is preferable to using a public instance field.)

# Resources

---

- Visual Studio's online documentation (MSDN) which you can reach from VS.Net by clicking the Help menu, under Contents. Under Development Tools and Languages, under C# you will find the C# Programming Guide and C# Reference with great information about C# syntax and features. Also available at [msdn.microsoft.com](http://msdn.microsoft.com)
- [www.CodePlex.com](http://www.CodePlex.com) - lots of information about .Net and related technologies
- [www.asp.net](http://www.asp.net) - a great site with lots of tutorials and examples on web technologies, primarily
- [www.CodeProject.com](http://www.CodeProject.com) - A great Visual Studio and .Net resource with sample code, tutorials and more.
- [msdn.microsoft.com/msdnmag](http://msdn.microsoft.com/msdnmag) - the MSDN magazine - lots of articles and sample code on .Net technologies
- [www.4GuysFromRolla.com](http://www.4GuysFromRolla.com) - great site on all things .Net
- [www.DatagridGirl.com](http://www.DatagridGirl.com) - specializing in advanced ASP.Net DataGrids.
- Books - there are lots of them out there. Anything from Microsoft Press, O'Reilly or APress tend to be good.
- Avoid "24 Hour" series books.
- Some of the WROX books (red cover.) can be disappointing.

# Homework

---

- Look up and read definitions for all Operators
- Look up and read definitions for all Access Modifiers
- Write a simple program using what you have learned