Contents

Introduction

Topic: .NET Framework and C#

Details: This presentation describes some elementary features of .NET Framework and C#.

Organization: The index that contains the chapters of this document is found in the first slide. On the top-left corner of every slide show the name of the chapter, sub-chapter etc. The titles are at top-right

.NET Basics

- The .NET Framework is a framework for developing and implementing software for personal computer, web etc.
- It was designed and is maintained by Microsoft Corporation.
- It came out around the year 2000, even though Microsoft started its development in early 90s.
- .NET has a rich collection of class library (called the Base Class Library) to implement GUI, query, database, web services etc.

.NET Basics

- Programs developed with .NET needs a virtual machine to run on a host. This virtual machine is called Common Language Runtime (CLR).
- Since the compiler doesn't produce native machine code, and its product is interpreted by the CLR, there's much security.
- .NET allows using types defined by one .NET language to be used by another under the Common Language Infrastructure (CLI) specification, for the conforming languages.

.NET Basics

- Any language that conforms to the Common Language Infrastructure (CLI) specification of the .NET, can run in the .NET run-time. Followings are some .NET languages.
  - Visual Basic
  - C#
  - C++ (CLI version)
  - J# (CLI version of Java)
  - A# (CLI version of ADA)
  - L# (CLI version of LISP)
  - IronRuby (CLI version of RUBY)

.NET Basics

Microsoft provides a comprehensive Integrated Development Environment (IDE) for the development and testing of software with .NET.

Some IDEs are as follows

- Visual Studio
- Visual Web Developer
- Visual Basic
- Visual C#
- Visual Basic

What is C#

C# is a general purpose object oriented programming language developed by Microsoft for program development in the .NET Framework.

It's supported by .NET's huge class library that makes development of modern Graphical User Interface applications for personal computers very easy.

It's a C-like language and many features resemble those of C++ and Java. For instance, like Java, it too has automatic garbage collection.

What is C#

It came out around the year 2000 for the .NET platform. Microsoft's Anders Hejlsberg is the principal designer of C#.

The "#" comes from the musical notation meaning C# is higher than C.

The current version (today's date is Nov 2, 2011) is 4.0 and was released on April, 2010

More information about C#, tutorial, references, support and documentation can be found in the Microsoft Developers Network website.

Types of Application

The product of the C# compiler is called the "Assembly". It's either a ".dll" or a ".exe" file. Both run on the Common Language Runtime and are different from native code that may also end with a ".exe" extension.

C# has two basic types of application.

- Windows From Application
  - This is GUI based, runs in a window of some sort
- Console Application
  - This application runs in the command prompt

A Typical and Trivial Program in C#

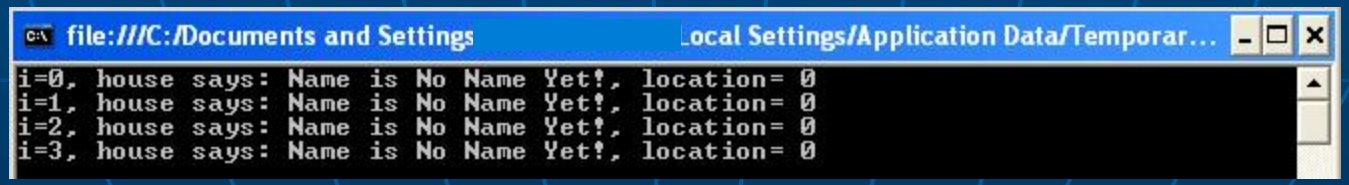
```
using System;
namespace typical_trivial{
    class House{
        private int location;
        protected string name;
        public House(){
            name = "No Name Yet!";
        }
        // every class inherits 'object' that has ToString()
        public override string ToString(){
            string disp = "Name is " + name + ", location= " +
location.ToString();
            return disp;
        }
    }
```

Continues to the next slide …

A Typical and Trivial Program in C#

… continuing from the previous slide

```
    class Program{
        static void Main(string[] args){
            House h = new House();
            for (int i = 0; i < 4; i++){
                System.Console.WriteLine("i={0}, house says:
{1}", i, h.ToString());
            }
            System.Console.Read();
        }
    }
}
```



Types

1. **Value type**
   1. Variable name contains the actual value
   2. int, double and other primitive types
   3. Structure, Enumeration, etc.
2. **Reference Type**
   1. Variable name contains the reference or pointer to the actual value in memory
   2. Array, derived from class Array
   3. Class, Interface, Delegate, String, etc.

Types

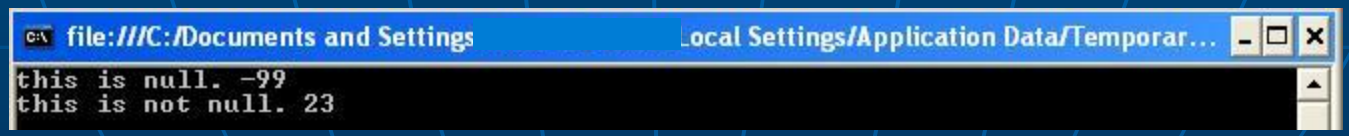
- The value types are derived from System.ValueType
- All types in C# are derived from System.Object which is also accessed by the alias keyword ‘object’
- This type hierarchy is called Common Type System (CTS)

Types

Nullable

The value types can't be assigned a null. To enable regular primitive value types to take a null value, C# uses nullable types using '?' with type name. Following example shows how.

```
int? a; a = null; int b; b = a ?? -99;
// the ?? operator picks -99 if null
System.Console.WriteLine("this is null. {0}",b);
a = 23; // not null
System.Console.WriteLine("this is not null. {0}", a ?? -99);
```

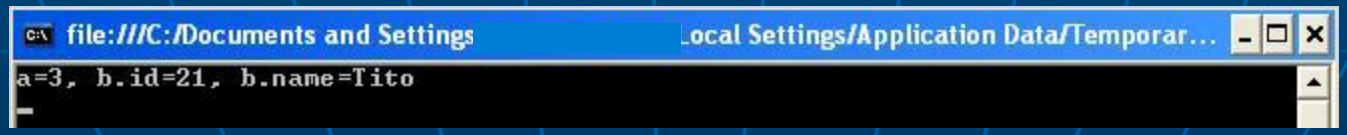


Types

Anonymous

Variables can be defined without an explicit name and to encapsulate a set of values. This is useful for C#'s Language Integrated Query (which will not be discussed in this presentation)

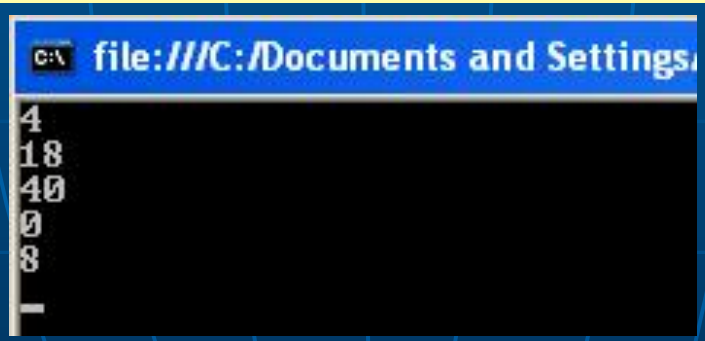
```
var a = 3; // the type is automatically inferred by compiler
var b = new { id = 21, name = "Tito" };
System.Console.WriteLine("a={0}, b.id={1}, b.name={2}", a,
       b.id, b.name);
```



Arrays

Following is an example of declaring and using a simple array.

```
int[] items = new int[]{5,19,41,1,9};
foreach (int i in items)
{
    System.Console.WriteLine("{0}\n", i-1);
}
```



Array

The ‘foreach’ ,’in’ keywords are used to provide read only (recommended) access to members of an array or any object implementing the IEnumerable interface (more about this is discussed in the section ‘Iterator’).

Properties

Properties are members of a class that allows for easy and simplified getters and setters implementation of its private field variables.

The next slide has an example.

Properties

```
class Client{
    private string name ;
    public string Name{
        get{
            return name;
        }
        set{
            name=value;
        }
    }
    static void Main(string[] args)
    {
        Client c = new Client();
        c.Name = "Celia";
        System.Console.WriteLine(c.Name);
        System.Console.ReadLine();
    }
}
```

Properties

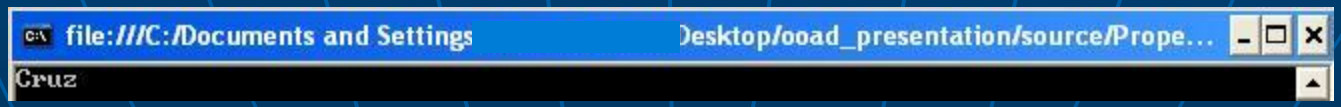
Automatically Implemented

C# also has a feature to automatically implement the getters and setter for you.

Users have direct access to the data members of the class.

Following example does the same thing as the previous example, but using automatically implemented properties

```
class Client2{
    public string Name { get; set; }
    static void Main(string[] args){
        Client2 c = new Client2();
        c.Name = "Cruz";
        System.Console.WriteLine(c.Name);
        System.Console.ReadLine();
    }
}
```



Indexers

Indexers allow a class to be used as an array. For instance the “[]” operator can be used and the ‘foreach’, ‘in’ keywords can also be used on a class that has indexers.

The internal representation of the items in that class are managed by the developer.

Indexers are defined by the following expression.

```
public int this[int idx]{
  get{/* your code*/}; set{/*code here*/};
}
```

Nested Classes

C# supports nested class which defaults to private.

```
class Program
{
    public class InsiderClass
    {
        private int a;
    }
    static void Main(string[] args)
    {
    }
}
```

Inheritance and Interface

A class can directly inherit from only one base class and can implement multiple interfaces.

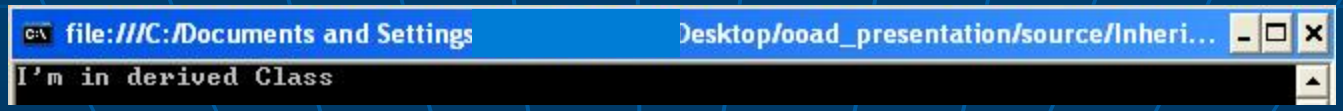
To override a method defined in the base class, the keyword ‘override’ is used.

An abstract class can be declared with the keyword ‘abstract’.

A static class is a class that is declared with the ‘static’ keyword. It can not be instantiated and all members must be static.

Inheritance and Interface

```
class BaseClass{
    public virtual void show(){
        System.Console.WriteLine("base class");}
}
interface Interface1{void showMe();}
interface Interface2{void showYou();}
class DerivedAndImplemented: BaseClass,Interface1,Interface2{
    public void showMe() { System.Console.WriteLine("Me!"); }
    public void showYou() { System.Console.WriteLine("You!"); }
    public override void show(){
        System.Console.WriteLine("I'm in derived Class");}
    static void Main(string[] args){
            DerivedAndImplemented de = new DerivedAndImplemented();
            de.show();
            System.Console.Read();}
}
```



Class Access and Partial

The class access modifiers are public, private, protected and internal. ‘internal’ is an intermediate access level which only allows access to classes in the same assembly.

The ‘partial’ keyword can be used to split up a class definition in to multiple location (file etc). Can be useful when multiple developers are working on different parts of the same class.

Delegates

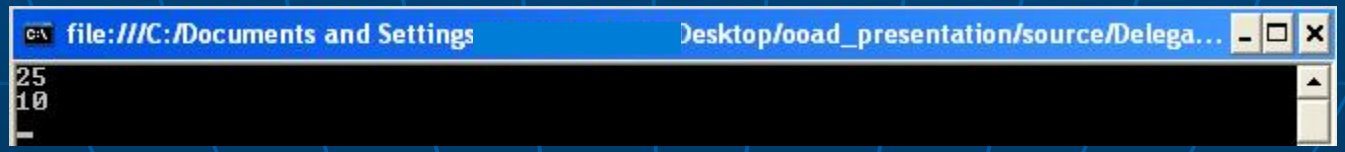
Delegates are types that describe a method signature. This is similar to function pointer in C.

At runtime, different actual methods of same signature can be assigned to the delegate enabling encapsulation of implementation.

These are extensively used to implement GUI and event handling in the .net framework (will be discussed later).

Delegates

```
class Program
{
    delegate int mydel(int aa);
    int myfunc(int a){ return a*a; }
    int myfunc2(int a) { return a + a; }
    static void Main(string[] args)
    {
        Program p=new Program();
        mydel d=p.myfunc; System.Console.WriteLine(d(5));
        d = p.myfunc2; System.Console.WriteLine(d(5));
        System.Console.Read();
    }
}
```

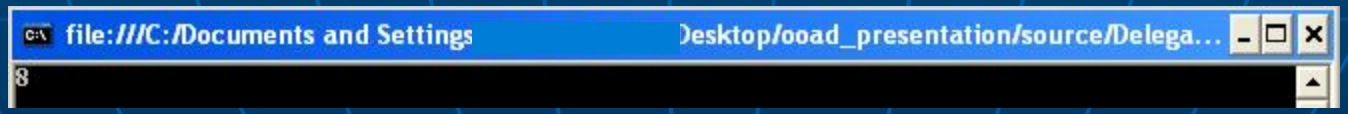


Delegates

Lambda Expression

In C#, implementors (functions) that are targeted by a delegate, can be created anonymously, inline and on the fly by using the lambda operator "=>".

```
class Program {
    delegate int mydel(int aa, int bb);
    static void Main(string[] args) {
        mydel d = (a, b) => a + 2 * b;
        // in above line, read a,b go to a+b*2 to evaluate
        System.Console.WriteLine(d(2,3));
        System.Console.Read();
    }
}
```



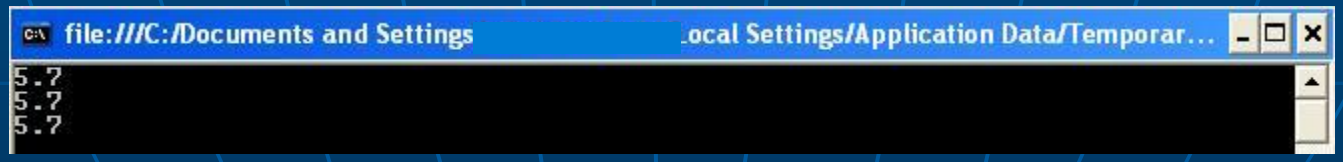
Generics

Generics are a powerful feature of C#. These enable defining classes and methods without specifying a type to use at coding time. A placeholder for type <T> is used and when these methods or classes are used, the client just simply has to plug in the appropriate type.

Used commonly in lists, maps etc.

Generics

```
class Genclass<T>{
    public void genfunc(int a, T b){
        for (int i = 0; i < a; i++){
            System.Console.WriteLine(b);
        }
    }
}
class Program{
    static void Main(string[] args){
        Genclass<float> p = new Genclass<float>();
        p.genfunc(3,(float)5.7);
        System.Console.Read();
    }
}
```



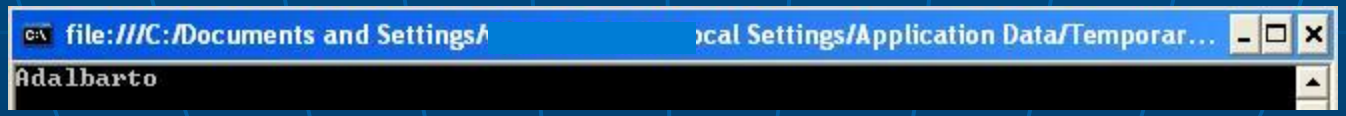
Object Initializer

Using automatically implemented properties (discussed earlier), object initializers allow for initializing an object at creation time without explicit constructors. It can also be used with anonymous types (discussed earlier).

The following slide has an example.

Object Initializer

```
class Client2
{
    public string Name { get; set; }
    static void Main(string[] args)
    {
        Client2 c = new Client2 {Name="Adalbarto"};
        // above is the object initializer
        System.Console.WriteLine(c.Name);
        System.Console.ReadLine();
    }
}
```

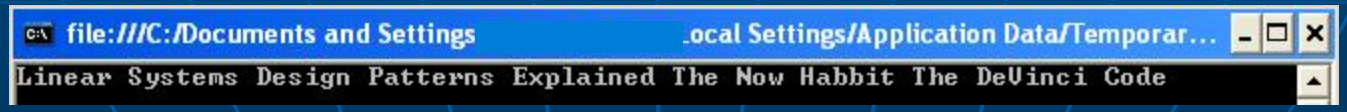


Iterator

Any class implementing the interface IEnumerable can be invoked by client code using the ‘foreach’, ‘in’ statements. The code loops through the elements of the class and provides access to its elements. This class defines the GetEnumerator method, where, individual elements are returned by the ‘yield’ keyword.

Iterator

```
    public class MyBooks : System.Collections.IEnumerable {
        string[] books = { "Linear Systems", "Design Patterns
Explained", "The Now Habbit", "The DeVinci Code" };
        public System.Collections.IEnumerator GetEnumerator() {
            for (int i = 0; i < books.Length; i++) {
                yield return books[i];
            }
        }
    }
    class Program {
        static void Main(string[] args) {
            MyBooks b = new MyBooks();
            foreach (string s in b) {
                System.Console.Write(s + " ");
            }
            System.Console.Read();
        }
    }
```



Structure

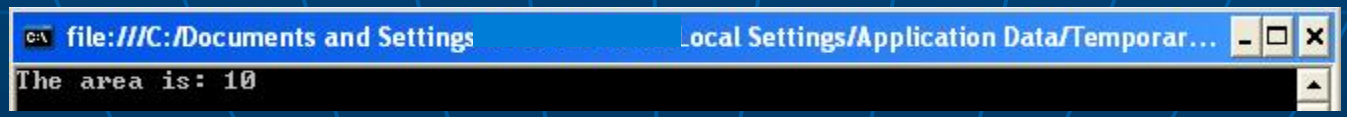
Structures are value types that can in some respect act similar to a class.

It has fields, methods, constructors (no argument constructors are not allowed) like a class.

Structures can't take part in inheritance, meaning that they can't inherit from a type and be a base from which other types can inherit.

Structure

```
struct Rectangle{
    public int length; public int width;
    public Rectangle(int length,int width){
        this.length=length; this.width=width;
    }
    public int getArea(){
        return length*width;
    }
}
class Program{
    static void Main(string[] args){
        Rectangle r=new Rectangle(2,5);
        System.Console.WriteLine("The area is: {0}",r.getArea());
        System.Console.Read();
    }
}
```



Namespace

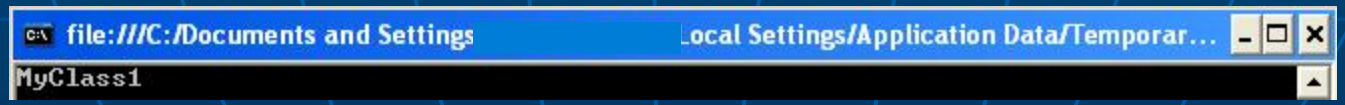
Names in C# belong to namespaces. They prevent name collision and offers a manageable code and libraries.

In the previous examples, ‘System’ is a namespace. System.Console.Write() is a method of a class defined in that namespace. So, the name of the namespace is put in front to fully qualify a name.

With the statement “using System;”, we can skip the System part and just write Console.Write().

Namespace

```
using System;
namespace space1{
    class MyClass1{
        public void show(){
            System.Console.WriteLine("MyClass1");
        }
    }
}
namespace space2{
    class Program{
        static void Main(string[] args){
            space1.MyClass1 c=new space1.MyClass1();
            c.show(); Console.Read();
        }
    }
}
```



Attribute

- Attributes add metadata to the code entities such as assembly, class, method, return value etc about the type.
- This metadata describes the type and it's members
- This metadata is used by the Common Runtime Environment or can also be used by client code.
- Attributes are declared in square brackets above the class name, method name etc.

Attribute

```
[Obsolete("Do not use this")]
public class Myclass {
    public void disp() {
        System.Console.WriteLine("..");
    }
}
class Program {
    [STAThread] // means thread safe for COM
    static void Main(string[] args) {
        Myclass mc = new Myclass(); mc.disp();
        System.Console.Read();
    }
}
```

The IDE

- Microsoft provides Visual Studio for the development of applications, web services etc using the .NET Framework with it's supported languages.
- Microsoft Visual C# Express is a free Microsoft product that can be used to develop C# applications for evaluation purposes.
- The examples provided in this presentation were developed using this software.

Other Miscellaneous Items

- Use of pointers: C# can be configured to allow pointer declaration and arithmetic.
- XML comments: It's possible to follow the XML comments syntax in code and the compiler will generate a documentation for you based of those comments and their location.
- Threading: It's possible to write multi-threaded programs using the System.Threading class library.
- C# allows easy integration to unmanaged code (outside of .NET) such as Win32Api, COM,C++ programs etc to it's own.

Other Miscellaneous Items

- C# allows editing the file system and the Windows System Registry.
- Exception: Exceptions can be handled using C#'s try, throw, catch.
- Collection Classes: These provide support for various data structures such as list, queue, hash table etc.
- Application Domain: The context in which the assembly is run is known as the application domain and is usually determined by the Common Language Runtime (it is also possible to handle this in code). This isolates individual programs and provides security.

GUI: Introduction

The .NET framework provides a class library of various graphical user interface tools such as frame, text box, buttons etc. that C# can use to implement a GUI very easily and fast.

The .NET framework also equips these classes with events and event handlers to perform action upon interacting with these visual items by the user.

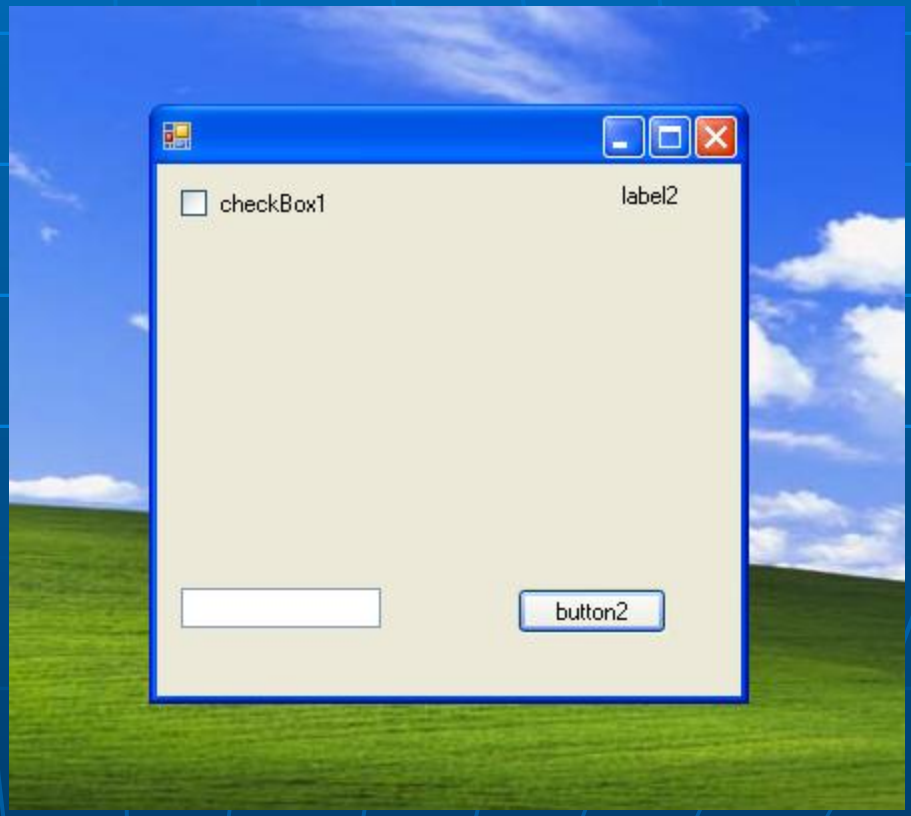
Visual Items

The following are some of the visual items.

- Form: displays as a window.
- Button: displays a clickable button
- Label: displays a label
- TextBox: displays an area to edit
- RadioButton: displays a selectable button

Visual Items

The containing window is called the “Frame”. Inside are some other GUI elements, such as Button, TextBox etc



Events

When anything of interest occurs, it's called an event such as a button click, mouse pointer movement, edit in a textbox etc.

An event is raised by a class. This is called *publishing* an event.

It can be arranged that when an event is raised, a class will be notified to handle this event, i.e. perform required tasks. This is called *subscribing* to the event. This is done via:

- Delegates or
- Anonymous function or
- Lambda expression.

These will be discussed shortly

Events

The .NET Framework has many built-in events and delegates for easily subscribing to these events by various classes.

For example, for Button class, the click event is called “Click” and the delegate for handler is called “System.EventHandler”. These are already defined in the .NET class library.

To tie the event with the handler, the operator “+=” is used. (Will be discussed shortly)

Then, when the Button object is clicked, that function will execute.

With Delegates

The following code segment shows the subscription of the event Button.Click by the method button_Click(…) which matches the System.EventHandler delegate signature.

```
Class Form1:Form{
   private System.Windows.Forms.Button button1;
   ///...
   Void init(){
      this.button1.Click += new System.EventHandler(this.button1_Click);
   }
   private void button1_Click(object sender, EventArgs e){
      button1.Text = "clicked1";
   }
}
```

With Lambda Expression

The following code segment shows the subscription of the event Button.Click by inline code which is defined using the lambda operator.

This program does the same thing as the last.

```
Class Form1:Form{
   private System.Windows.Forms.Button button1;
   ///...
   Void init(){
      // the arguments a,b are just to satisfy the delegate signature.
      // they do nothing useful in this simple example.
      this.button1.Click += (a,b) => { this.button1.Text = "clicked1"; };
   }
}
```

With Anonymous Methods

An anonymous method is declared with the keyword “delegate”. It has no name in source code level.

After the delegate keyword, the arguments need to be put in parenthesis and the function body needs to be put in braces.

It is defined in-line exactly where it’s instance is needed.

Anonymous methods are very useful in event programming in C# with .NET Framework.

With Anonymous Methods

The following example does the same thing as the previous two examples but uses anonymous methods to handle that event.

```
Class Form1:Form{
   private System.Windows.Forms.Button button1;
   ///...
   Void init(){
      this.button1.Click += delegate(object oo, System.EventArgs ee) {
         this.button1.Text = "clicked1";
      };
   }
}
```

Demo

For this section, please see the attached video demonstration of the following.

- Basic use of Visual C# IDE
- Showing of how easily GUI can be created
- Creation of a simple web browser

Conclusion

This presentation described in short some interesting and important features of the .NET and C#.

However, the .NET and C# are not without their trade offs such as the followings.

- The .NET currently doesn't support optimized Single Instruction Multiple Data (SIMD) support for parallel data processing.
- Since it's run in a virtual machine, the demands on system resources are higher than native code of comparable functionality.

Conclusion

Overall, .NET and C# are very robust, flexible, object oriented, with large library support, secure means of software design and development.

Some of the features that were discussed in this presentation, are as follows

- Virtual Machine provides security and isolation
- Common Runtime Infrastructure enables any language to adapt to the run time

Conclusion

- The Common Language Infrastructure and Common Type System makes it possible for multiple languages to use each others defined types and libraries.
- It has extensive class library for easily developing GUI and web application.
- C# has many very nice and flexible features such as partial class/method, nullable type, anonymous method, indexers, generics, iterators, properties etc.

The End