Agenda

- Iterators
- Partial Types
- Anonymous Methods
- Delegate Inference
- Nullable Types
- Property and Index Visibility
- Static Classes
- Global Namespace Qualifier
- Inline Warning
- Extension methods

Agenda (cont.)

- Automatic Properties
- Object Initializers
- Collection Initializers
- Anonymous Types
- Dynamic Lookup
- LamdaExpressions

Iterators

- An iterator is a method, get accessor, or operator that enables you to support foreach iteration in a class or struct without having to implement the entire IEnumerable interface.
  - Instead, you provide just an iterator, which simply traverses the data structures in your class.
  - When the compiler detects your iterator, it will automatically generate the Current, MoveNext, and Dispose methods of the IEnumerable interface.
- Iterators are especially useful with collection classes, providing an easy way to iterate non-trivial data structures such as binary trees.

Iterators Overview

- An iterator is a section of code that returns an ordered sequence of values of the same type.
- An iterator can be used as the body of a method, an operator, or a get accessor.
- The iterator code uses the **yield return** statement to return each element in turn. **yield break** ends the iteration:
  - The **yield** keyword is used to specify the value, or values, returned.
  - When the **yield return** statement is reached, the current location is stored.
  - Execution is restarted from this location the next time the iterator is called.
- Multiple iterators can be implemented on a class:
  - Each iterator must have a unique name just like any class member, and can be invoked by client code in a foreach statement as follows: foreach(int x in SampleClass.Iterator2){}.
- The return type of an iterator must be IEnumerable or IEnumerator.

Iterators Implementation

```
using System;
using System.Collections;
public class Persons : IEnumerable
{
  string[] names;
  public Persons(params string[] nameParam)
  {
     names = new string[nameParam.Length];
     nameParam.CopyTo(names, 0);
  }
  public IEnumerator GetEnumerator()
  {
     foreach (string nameStr in names)
     {
        yield return nameStr;
     }
  }
}
class TestProgram
{
  static void Main(string[] args)
  {
    Persons arrPersons = new Persons("Jack", "Jill", "Mathew");
    foreach (string name in arrPersons)
       Console.WriteLine(name);
  }
}
```

Output

```
Jack
Jill
Mathew
```

Partial Types

- C# 4.0 allows you to split the definition and implementation of a class or a struct, interface, and method over two or more Source files.
  - You can put one part of a class in one file and another part of the class in a different file by using the new **partial** keyword.
- Each source file should contain a section of the class or method definition. When you compile the application, the compiler combines all the source files.
- Example: MyPartialClass1.cs

```
public partial class MyClass
{
    public void Method1() {...};
}
```

MyPartialClass2.cs

```
public partial class MyClass
{
    private string myName;
    public void Method2() {...};
}
```

  - Note that both **MyPartialClass1.cs** and **MyPartialClass2.cs** contain the code for the same class MyClass.

Anonymous Methods (1 of 3)

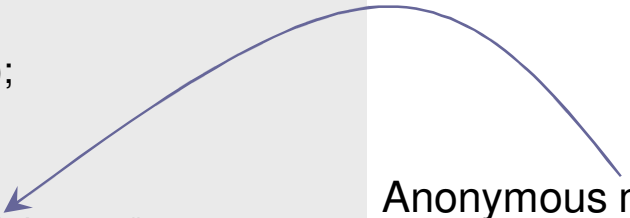
- In C#, you are sometimes forced to define a method just for the sake of using a delegate:
  - In such cases, there is no need for multiple targets, and the code involved is often relatively short and simple.
  - Anonymous methods is a new feature in C# 4.0 that lets you define an anonymous (that is, nameless) method called by a delegate.
- Example:

```
class SomeClass
{
    delegate void SomeDelegate();
    public void InvokeMethod()
    {
            SomeDelegate del = new SomeDelegate(SomeMethod);
            del();
    }
    void SomeMethod()
    {
            Console.Write("Hello");
    }
}
```

Anonymous Methods (2 of 3)

- In C# 4.0, the same can be done using the anonymous method:

```
class SomeClass
{
    delegate void SomeDelegate();
    public void InvokeMethod()
    {
            SomeDelegate del = delegate()
            {
                Console.Write("Hello");
            };
            del();
    }
}
```

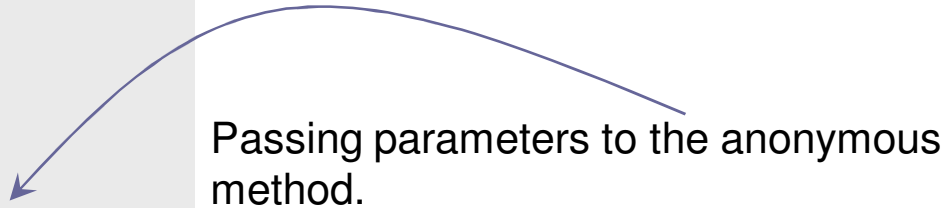


Anonymous method (defined in-line).

Anonymous Methods (3 of 3)

- Parameters can also be passed to the anonymous methods.
- When defining an anonymous method with parameters, you define the parameter types and names after the delegate keyword just as if it were a conventional method:

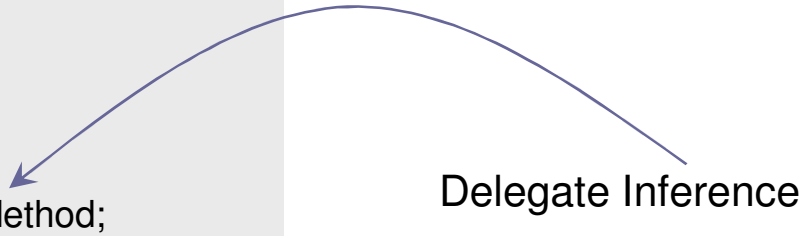
```
class SomeClass
{
    delegate void SomeDelegate(string str);
    public void InvokeMethod()
    {
            SomeDelegate del = delegate(string str)
            {
                Console.Write(str);
            };
            del("Hello");
    }
}
```



Delegate Inference

- Delegate inference allows you to make a direct assignment of a method name to a delegate variable, without wrapping it first with a delegate object.
- For example:

```
class SomeClass
{
    delegate void SomeDelegate();
    public void InvokeMethod()
    {
            SomeDelegate del = SomeMethod;
            del();
    }
    void SomeMethod()
    {
            ...
    }
}
```



Nullable Types

- Nullable types represent value-type variables that can be assigned the value of **null**.
  - Nullable types can represent the normal range of values for its underlying value type, plus an additional **null** value.
  - For example, a Nullable<Int32>, pronounced "Nullable of Int32," can be assigned any value from -2147483648 to 2147483647, or it can be assigned the **null** value.
- Nullable types are assigned as follows:
  - int? num = null;
- Use the System.Nullable.GetValueOrDefault property to return either the assigned value, or the default value for the underlying type if the value is null.
- Use the HasValue and Value read-only properties to test for null and retrieve the value.

Nullable Types (cont.)

```
class NullableExample
{
    static void Main()
    {
        int? num = null;
        if (num.HasValue == true)
        {
            System.Console.WriteLine("num = " + num.Value);
        }
        else
        {
            System.Console.WriteLine("num = Null");
        }
        //y is set to zero
        int y = num.GetValueOrDefault();
        // num.Value throws an InvalidOperationException if num.HasValue is false
        try
        {
            y = num.Value;
        } catch (System.InvalidOperationException e)
        {
            System.Console.WriteLine(e.Message);
        }
    }
}
```

Nullable type

Output

```
num = Null
Nullable object must have a value.
```

Property and Index Visibility

- C# 4.0 allows you to specify different visibility for the get and set accessors of a property or an indexer.
- The visibility qualifier you apply on the set or the get can only be a stringent subset of the visibility of the property itself:
  - In other words, if the property is public, then you can specify internal, protected, protected internal, or private.
  - If the property visibility is protected, you cannot make the get or the set public.
- The visibility can be specified only for the get or the set, but not both.
- Example:

```
public class MyClass
{
        string[] m_Names;
        public string this[int index]
        {
                get { return m_Names[index]; }
                protected set { m_Names[index] = value; }
        }
        //Rest of the class
}
```

A different visibility for the set method.

Static Classes

- In previous versions of .NET, to prevent developers from instantiating objects of your class, you provided a private default constructor on a sealed class.
- In C# 4.0, use the 'static' keyword to make static classes.
  - The C# 4.0 compiler does not allow you to add a non-static member to the static class and will not allow you to create an instance of the static class.
  - Also, you cannot derive from a static class.
- For example:

```
public static class MyClassFactory
{
    public static T CreateObject<T>()
    {
        ...
    }
}
```

Using the 'static' keyword, the class can be made a static class.

Global Namespace Qualifier

- In C# 4.0 it is possible to have a nested namespace with a name that matches another global namespace:
  - However, the older version .NET compiler will have trouble resolving the namespace reference.
- C# 4.0 allows you to use the global namespace qualifier global:: to indicate to the compiler that it should start its search at the global scope.

- Example:

```
namespace MyApp
{
    class MyClass
    {
        public void MyMethod()
        {
            global::MyClass obj = new global::MyClass();
            obj.MyMethod();
        }
    }
}
public class MyClass
{
    public void MyMethod()
    {
        Trace.WriteLine("Hello");
    }
}
```

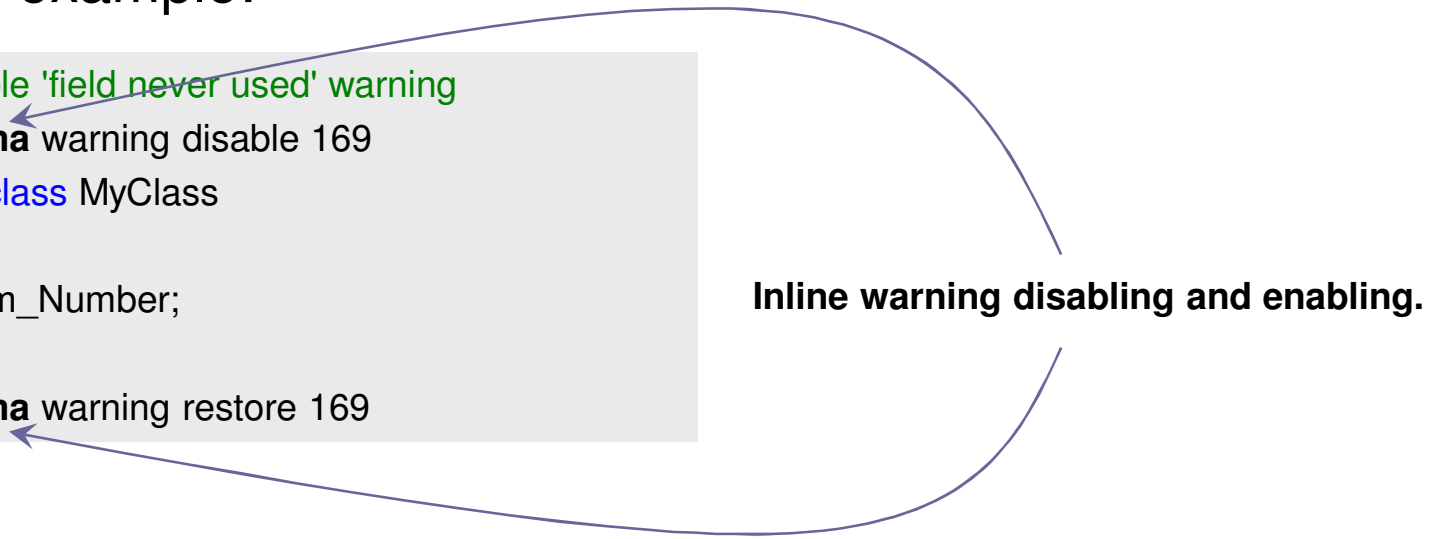
Using the global namespace identifier, the compiler can be instructed to start the search from the global scope.

Traces "Hello" instead of recursion.

Inline Warning

- C# previous version .NET allows you to disable specific compiler warnings using project settings or by issuing command-line arguments to the compiler:
  - The problem here is that this is a global suppression, and as such, suppresses warnings that you still want.
  - C# 4.0 allows you to explicitly suppress and restore compiler warnings using the #pragma warning directive.
- For example:

```
// Disable 'field never used' warning
#pragma warning disable 169
public class MyClass
{
    int m_Number;
}
#pragma warning restore 169
```



Inline warning disabling and enabling.

Extension Methods

- Give us the capabilities to asses our own custom methods to datatypes without deriving from the base class.
- Extension methods are special static methods which act like instance methods on extended types.

```
Using System;
Namespace DemoExtensionMethos
{
Static class ExtenstionString
{
        public static int ToInteger(string val)
        {
        return Convert.ToInt32(val);
        }
}
}
```

```
Using System;
Using DemoExtenstionDemo
{
        class Demo
        {
        static void Main(string[] arg)
        {
                string Ex="100";
                int a=ex.ToInteger();
                Console.WriteLinea();
        }
        }
}
```

Automatic Properties

- Allows you to type less code and still get a private field and its public getter and setter.
- Compiler will generate the private field and public setter and getter for you.

```
//Employee class with getter and setter
Class Employee
{
        private int Emp_Code;
        private string Emp_Name;
        public int EmployeeCode
        {
                get{return Emp_code;}
                set{Emp_Code=value;}
        }
        public string EmpolyeeName
        {
                get{return Emp_Name;}
                set{Emp_name=value;}
        }
}
```

```
//Employee class with automatic properties
Class Employee
{
        private int Emp_Code;
        private string Emp_Name;
        public int EmployeeCode
        {
                get;
                set;
        }
        public string EmpolyeeName
        {
                get;
                set;
        }
}
```

Object Initializer

- Object Initializer allows you to pass in named values for each of the public properties that will be used to initialize the object.
- Allows you to pass in any named public property to the constructor of the class.
- This feature removes the need to create multiple overloaded constructors using different parameter lists.

```
//class with Automatic Properties
public class Employee
{
        public int EmployeeCode{ get; set;}
        public string EmployeeName{ get; set;}
        public int EmployeeAge{ get; set;}
}
```

```
//With instances and initializing an object
Employee emp=new Employee
{
        EmployeeCode=100,
        EmployeeName="Raj",
        EmployeeAge=25
} ;
```

Collection Initializes

- Collection Initializes allow you to create a collection and initialize it with a series of objects in a single statement.

```
//Combining object and collection initializers
List<Employee> emList=new List<Employee>
{
        new Employee{Employeecode=100,Employeename=“Rajesh”},
        new Employee{Employeecode=101,Employeename=“Devid”},
        new Employee{Employeecode=102,Employeename=“Sriram”},
};
```

```
//Combining object and collection initializers
List<Employee> empList=new List<Employee>();
empList.Add(new Employee(101,”Ramesh”,23));
empList.Add(new Employee(102,”Ram”,23));
empList.Add(new Employee(103,”Suresh”,23));
```

Anonymous Types

- Anonymous Types use the C# compiler to automatically create types based on the data that you want to store in them.
- Anonymous types do not have identifiers.
- No type name is specified after the new Keyword.

```
//Employee type can be instantiated as follows
Employee emp=new Employee
{
        EmployeeCode=111;
        EmployeeName="Raghav";
        EmployeeAge=26;
};
```

```
//Anonymous type as follows
Var emp=new
{
        EmployeeCode=111;
        EmployeeName="Raghav";
        EmployeeAge=26;
}
```

Dynamic Lookup

- The variables do not have fixed types.
- Dynamic keyword needs to be used to define variables.
- In most cases, it functions like it has type object.
- Dynamic language runtime(DLR) is a new API. It provides the infrastructure that supports the dynamic type in C#.

```
//dynamic variables declarations
dynamic dy1=100;
dynamic dy2=“Hello”;
dynamic dy3=System.DataTime.Today;
```

Lambda Expressions

- A lambda expression is an unnamed method written in place of a delegate instance.
  - An *expression tree*, of type Expression<T>, representing the code inside the lambda expression in a traversable object model.
- In the following example, Multiply is assigned the lambda expression x = > x * x:

```
delegate int Example (int j);
class Test
{
 static void Main( )
 {
     Example Multiply = x => x * x;
    Console.WriteLine (Multiply(2)); // 4
}
}
```