.NET Coding Best Practices

Lesson Objectives

In this lesson, you will learn:

- C# Language Coding Guidelines and Best Practices
- ADO.NET Coding Guidelines and Best Practices
- ASP.NET Coding Guidelines and Best Practices
- WCF Coding Guidelines and Best Practices

Overview

A Comprehensive coding standard is essential for a successful product delivery.

The standard helps in enforcing best practices and avoid pitfalls, and makes knowledgeable dissemination across the team easier

This course addresses some fundamental coding techniques and provide a collection of coding practices.

The coding techniques are primarily those that improve the readability and maintainability of code, where are the programming practices are mostly performance enhancements.

.NET Design Rules

Abstract types should not have constructors

- Public constructors for abstract types do not make sense because you cannot create instances of abstract types.

Avoid empty interfaces

- Interfaces are meant to contain members that specify a set of behaviors. To mark or label a class, use an attribute instead of an empty interface.

Avoid out parameters

- Using out parameters might indicate a design flaw. Although there are legitimate times to use out parameters, their use frequently indicates a design that does not conform to the design guidelines for managed code.

Declare types in namespaces

- A type should be defined inside a namespace to avoid duplication.

.NET Design Rules

Do not catch general exception types

- You should not catch Exception or SystemException.
- Catching generic exception types can hide run-time problems from the library user, and can complicate debugging.
- You should catch only those exceptions that you can handle gracefully.

Do not declare protected members in sealed types

- Sealed types cannot be extended, and protected members are only useful if you can extend the declaring type.
- Sealed types should not declare protected members.

Do not declare virtual members in sealed types

- Sealed types cannot be extended, and virtual members are only useful if you can extend the declaring type.

.NET Design Rules

Do not overload operator equals on reference types

- Most reference types, including those that override System.Object.Equals, do not override the equality operator (==).
- Most languages provide a default implementation of this operator.

Do not pass types by reference

- Although there are legitimate times to use reference parameters, such use frequently indicates a design that does not conform to the design guidelines for managed code.

Implement IDisposable correctly

- All IDisposable types should implement the Dispose pattern correctly.

Mark attributes with AttributeUsageAttribute

- The AttributeUsage attribute specifies the targets that are valid for the attribute (see System.AttributeTargets), whether an attribute can appear on a target more than once, and whether the attribute is inheritable.

.NET Design Rules

Properties should not be write only

- Write-only properties usually indicate a flawed design.

Use properties where appropriate

- Properties should be used instead of Get/Set methods in most situations.

Naming Guidelines

Of all the components that make up the coding standard, naming standards are the most visible and arguably the most important.

Namespace naming guidelines

Use meaningful namespaces such as the product name or the company name.

Prefixing namespace names with a company name or other well established brands avoids the possibility of two published namespaces having the same name.

- e.g. Microsoft.Office

Class Naming Guidelines

Use a noun or noun phrase to name a class

- Use Pascal Case
- Do not use type prefix, such as C or Class, on a class name.
- E.g use the class name **FileStream** rather than **CFileStream** or **ClsFileStream**
- Do not use underscore character.

Where appropriate, use a compound word to name a derived class

- The second part of the derived class's name should be the name of the base class.
- E.g. ApplicationException : Exception

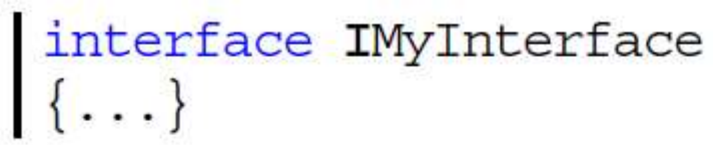
Interface naming guidelines

Name interfaces with nouns or noun phrases, or adjectives that describe behaviors.

- For example, the interface name *Icomponent* uses a descriptive noun.
- The interface name ICustomAttributeProvider uses a noun phrase
- The name IPersistable uses an adjective.

Use Pascal case

Prefix interface names with I to indicate that the type is an interface.



```
interface IMyInterface
{...}
```

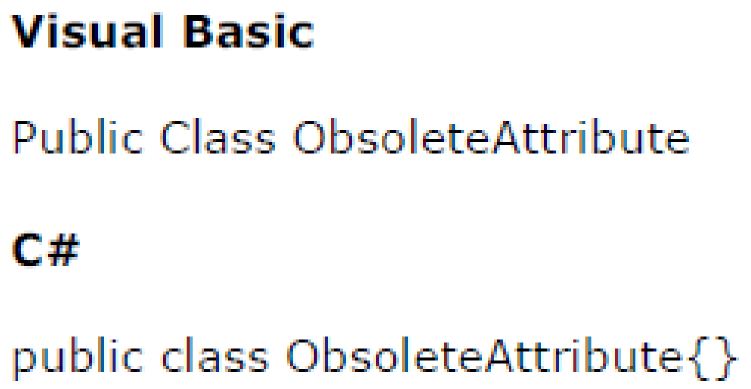
Do not use underscore character

Interface naming guidelines

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

Attribute

One should always add the suffix Attribute to custom attribute classes.



Visual Basic

```
Public Class ObsoleteAttribute
```

C#

```
public class ObsoleteAttribute{}
```

Enumeration

The enumeration (Enum) value type inherits from the Enum class.

Use Pascal Case for Enum types and value names

Do not use Enum suffix on Enum type names

```
// Correct
public enum Color
{
    Red,
    Green,
    Blue,
    Yellow,
    Magenta,
    Cyan
}
```

Static Field

Use Nouns, noun phrases, or abbreviations of nouns to name static fields.

Use Pascal Case

```
// Correct
public class Account
{
    public static string BankName;
    public static decimal Reserves;

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}
    public DateTime DateClosed {get; set;}
    public decimal Balance {get; set;}
```

Parameter Naming Guidelines

Use descriptive parameter names.

- Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.

Use Camel Case

Use names that describe a parameter's meaning rather than describe a parameter's type.

```
public class UserLog
{
    public void Add(LogEvent logEvent)
    {
        int itemCount = logEvent.Items.Count;
        // ...
    }
}
```

Type and Method Naming Guidelines

Use Pascal casing for type and method names and constants:

```
public class SomeClass
{
    const int DefaultSize = 100;
    public void SomeMethod()
    {}
}
```

Use Camel casing for local variable names and method arguments

```
void MyMethod(int someNumber)
{
    int number;
}
```

Method Naming

Name methods using Verb-object pair, such as ShowDialog ()

Some Examples:

- RemoveAll ()
- GetCharArray ()
- Invoke ()

Property Naming

Use a noun or noun phrase to name properties

Use Pascal case

Do not use Hungarian notation

```
// Correct
public class Account
{
    public static string BankName;
    public static decimal Reserves;

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}
    public DateTime DateClosed {get; set;}
    public decimal Balance {get; set;}
```

Event Naming

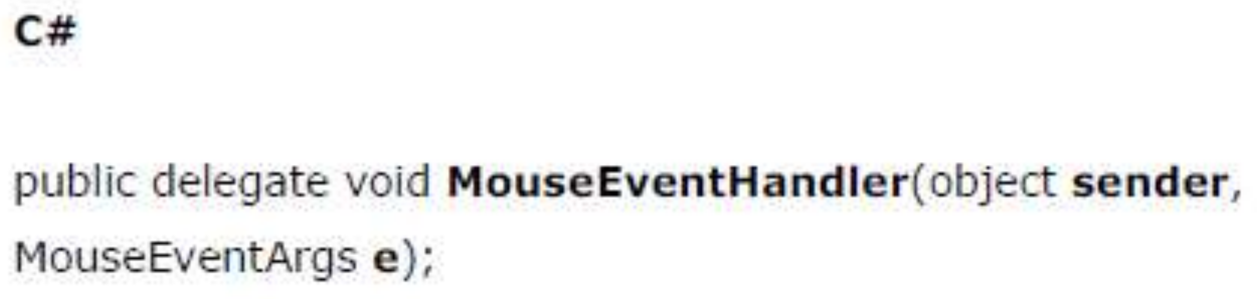
Use an EventHandler suffix on event handler names

Specify two parameters named *sender* and *e*.

- The sender parameter represents the object that raised the event.
- The state associated with the event is encapsulated in an instance of an event class named.

Name an event argument class with the EventArgs suffix

Consider naming events with a verb


C#

```
public delegate void MouseEventHandler(object sender,
MouseEventArgs e);
```

UI Elements

- **Naming Conventions for UI Elements:**
  - Use Hungarian notation while naming UI controls. Use appropriate prefix for the UI elements to identify them from the rest of the variables
  - Two different approaches are recommended:
    - Use a common prefix (ui) for all UI elements. This will help to group all the UI elements together and easy to access all of them from the intellisense
    - Use appropriate prefix for each of the UI element

Standards

Control	Prefix
Label	lbl
TextBox	txt
DataGrid	dtg
Button	btn
ImageButton	imb
Hyperlink	hlk
DropDownList	ddl

Standards

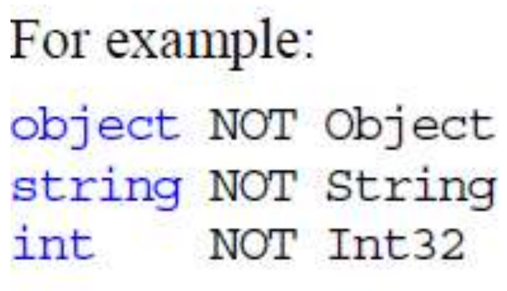
Control	Prefix
ListBox	lst
DataList	dtl
Repeater	rep
Checkbox	chk
CheckBoxList	cbl
RadioButton	rdo
RadioButtonList	rbl

Standards

Control	Prefix
Image	img
Panel	pnl
PlaceHolder	phd
Table	tbl
Validators	val

General Guidelines (contd.)

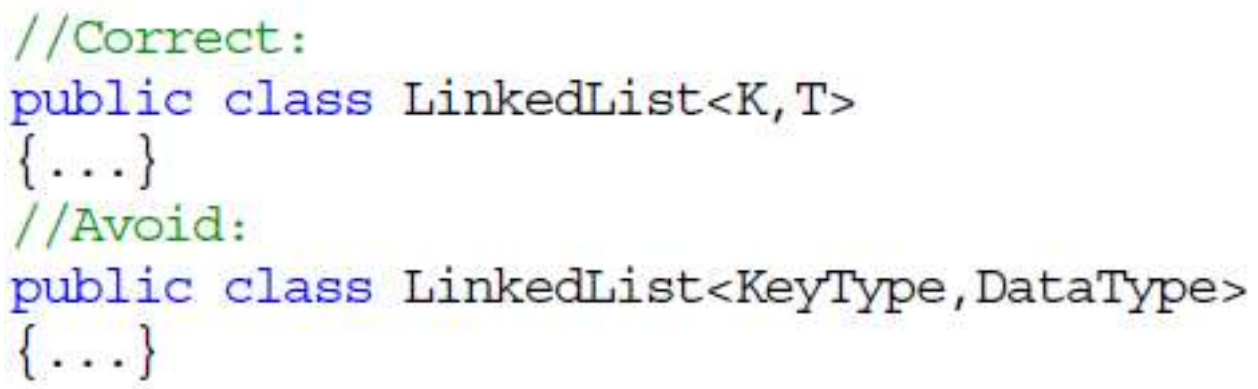
Always use C# predefined types rather than the aliases in the System namespace.



For example:

```
object NOT Object
string NOT String
int    NOT Int32
```

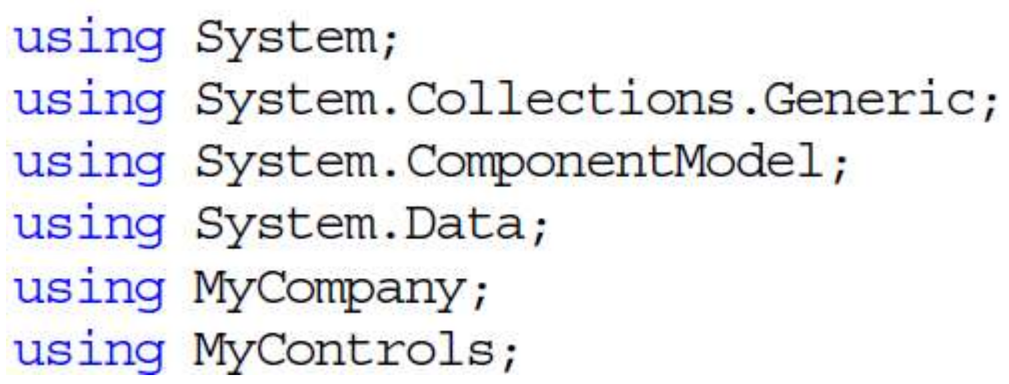
With generics, use capital letters for types. Reserve suffixing Type when dealing with the .NET type Type.



```
//Correct:
public class LinkedList<K,T>
{...}
//Avoid:
public class LinkedList<KeyType,DataType>
{...}
```

General Guidelines (contd.)

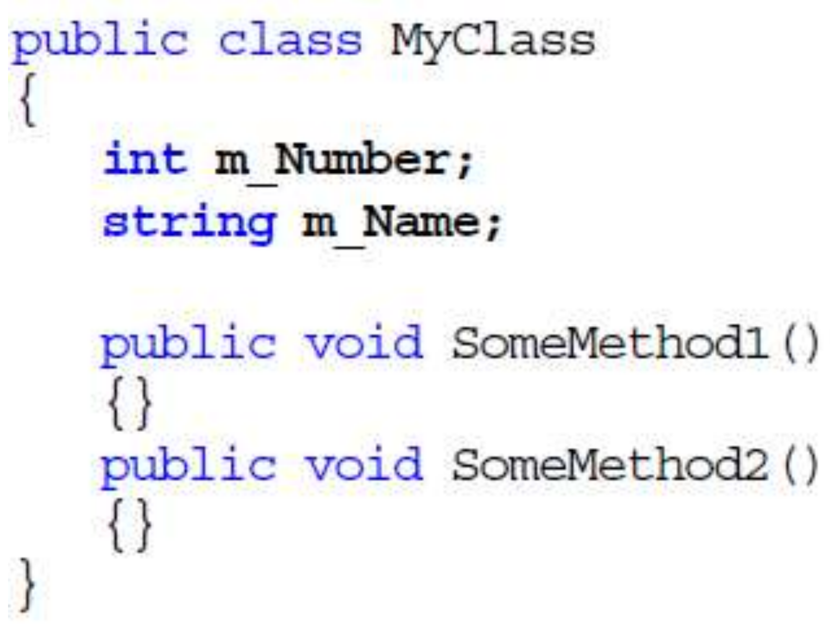
Group all framework namespaces together and put custom or third party namespaces underneath.



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using MyCompany;
using MyControls;
```

General Guidelines (contd.)

All member variables should be declared at the top, with one line separating them from the properties or methods.



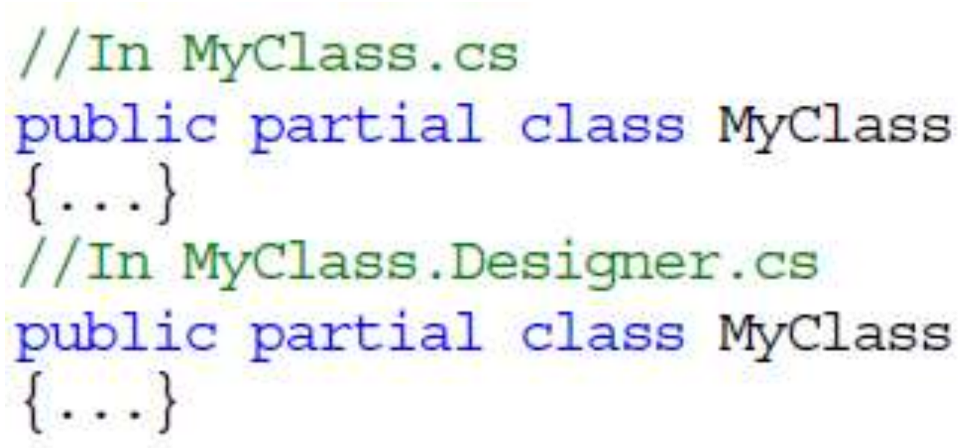
```
public class MyClass
{
    int m_Number;
    string m_Name;

    public void SomeMethod1()
    {}
    public void SomeMethod2()
    {}
}
```

General Guidelines (contd.)

Declare a local variable as close as possible to its first use.

A file name should reflect the class it contains.

When using partial types and allocating a part per file, name each file after the logical part that part plays.



```
//In MyClass.cs
public partial class MyClass
{...}
//In MyClass.Designer.cs
public partial class MyClass
{...}
```

Anonymous methods

Always place an open curly brace ({) in a new line.

With anonymous methods, mimic the code layout of a regular method, aligned with the delegate declaration

```
delegate void SomeDelegate(string someString);
//Correct:
void InvokeMethod()
{
   SomeDelegate someDelegate = delegate(string name)
                               {
                                  MessageBox.Show(name);
                               };
   someDelegate("Juval");
}

//Avoid
void InvokeMethod()
{
   SomeDelegate someDelegate = delegate(string name){MessageBox.Show(name);};
   someDelegate("Juval");
}
```

General Guidelines (contd.)

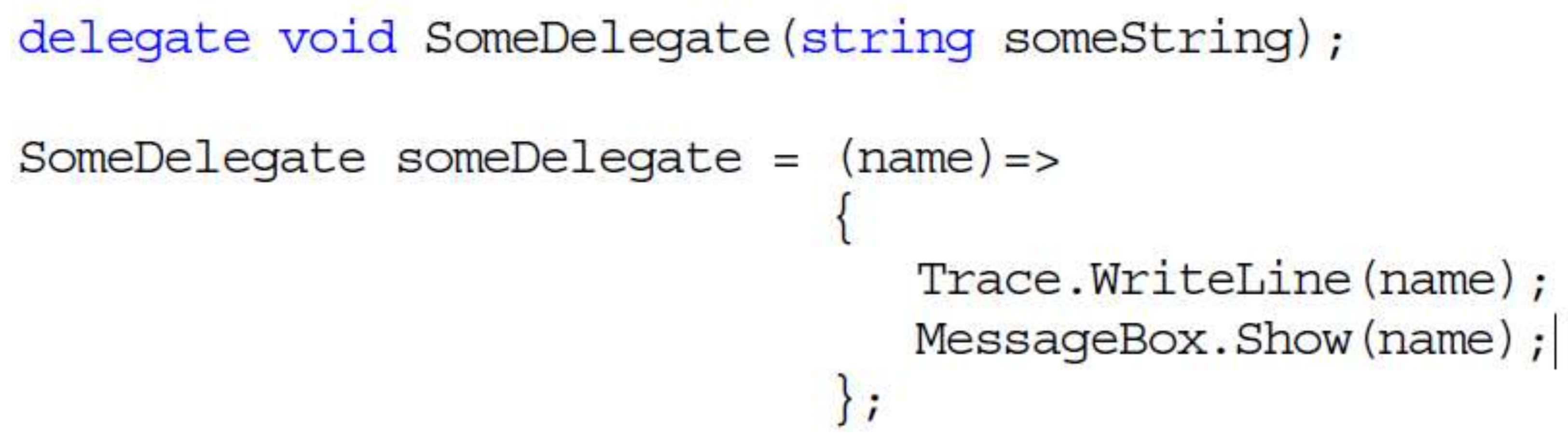
Use empty parentheses on parameter-less anonymous methods.

Omit the parentheses only if the anonymous method could have been used on any delegate.

```
delegate void SomeDelegate();
//Correct
SomeDelegate someDelegate1 = delegate()
                             {
                                MessageBox.Show("Hello");
                             };
//Avoid
SomeDelegate someDelegate1 = delegate
                             {
                                MessageBox.Show("Hello");
                             };
```

General Guidelines (contd.)

With Lambda Expressions, mimic the code layout of a regular method, aligned with the delegate declaration. Omit the variable type and rely on type inference, yet use parentheses:



```
delegate void SomeDelegate(string someString);

SomeDelegate someDelegate = (name)=>
                            {
                              Trace.WriteLine(name);
                              MessageBox.Show(name);
                            };
```

General Guidelines (contd.)

Only use in-line Lambda expressions when they contain a single simple statement. Avoid multiple statements that require a curly brace or a return statement with in-line expressions. Omit parenthesis:

```
delegate void SomeDelegate(string someString);

void MyMethod(SomeDelegate someDelegate)
{...}

//Correct:
MyMethod(name=>MessageBox.Show(name));

//Avoid
MyMethod((name)=>{Trace.WriteLine(name);MessageBox.Show(name);});
```

Usage Rules

Call GC.SuppressFinalize correctly

- Call GC.SuppressFinalize to suppress finalization of your instance once Dispose has been called.
- Objects that implement IDisposable should call SuppressFinalize from the IDisposable.Dispose method to prevent the garbage collector from calling Object.Finalize on an object that does not require it.

Disposable types should declare finalizer

- If a type implements a Dispose method and has unmanaged resources, it should provide a Finalize method in case Dispose is not explicitly called.
- The runtime calls the Finalize method or destructor of an object prior to reclaiming its managed resources in a process known as garbage collection.
- The Finalize method should free any unmanaged resources before they become inaccessible.

Usage Rules

Do not decrease inherited member visibility

- It is incorrect to override a public method from an inherited class with a private implementation, unless the type is sealed or the method is marked final.
- It is considered bad form to hide a method signature halfway down an inheritance tree.

Test for empty strings using string length

- To test for empty strings, check if String.Length is equal to zero.
- Constructs such as "".Equals(someString) and String.Empty.Equals(someString) are less efficient than testing the string length.
- Replace these with checks for someString.Length == 0.

Coding Practices

Avoid putting multiple classes in a single file.

A single file should contribute types to only a single namespace.

- Avoid having multiple namespaces in the same file.

Avoid files having more that 500 lines

Avoid methods with more than 200 lines

Avoid methods with more than 5 arguments.

- Use structures for passing multiple arguments.

Coding Practices

Do not manually edit any machine-generated code.

Avoid comments that explain the obvious.

- Code should be self-explanatory.
- Good code with readable varible names and method names should not require comments

Document only operational assumptions, algorithm insights and so on.

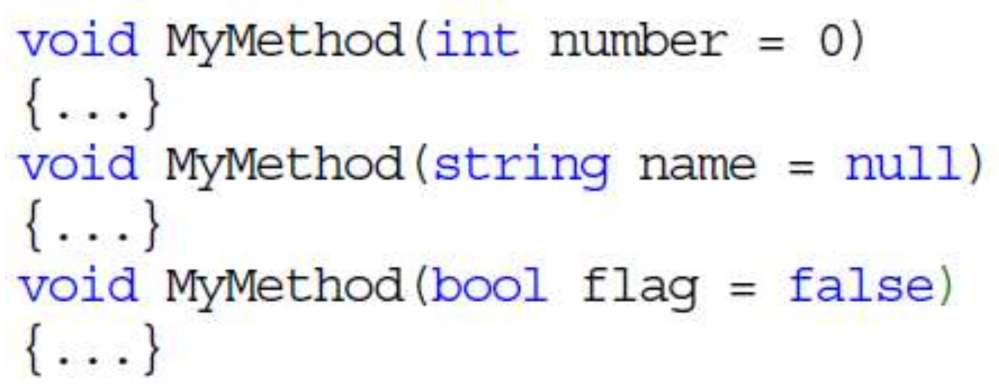
Coding Practices

In general, prefer overloading to default parameters:

```
//Avoid:
class MyClass
{
   void MyMethod(int number = 123)
   {...}
}
//Correct:
class MyClass
{
   void MyMethod()
   {
      MyMethod(123);
   }
   void MyMethod(int number)
   {...}
}
```

Coding Practices

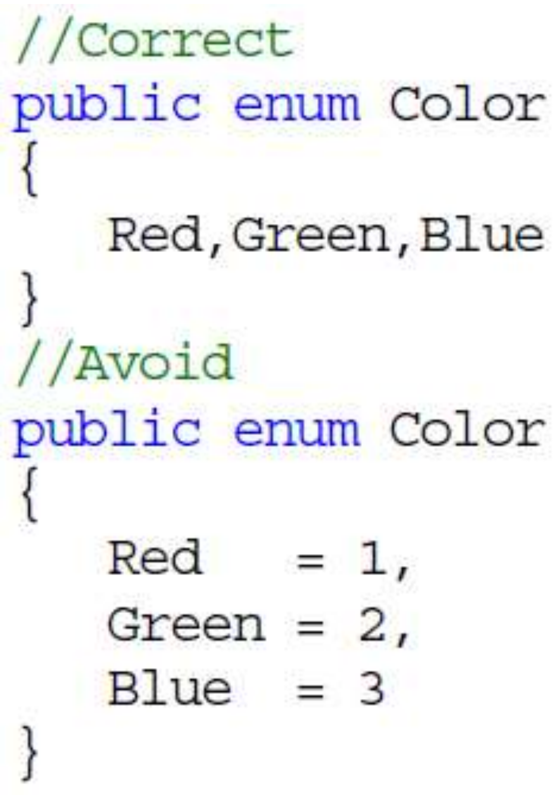
When using default parameters, restrict them to natural immutable constants such as null, false or 0:



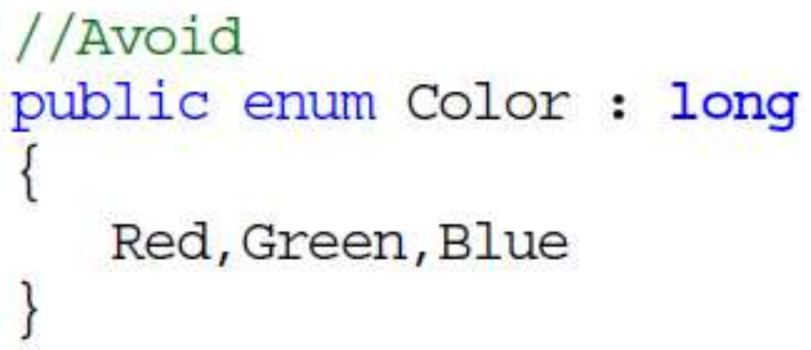
```
void MyMethod(int number = 0)
{...}
void MyMethod(string name = null)
{...}
void MyMethod(bool flag = false)
{...}
```

Coding Practices

Avoid providing explicit values for enums, and specyfing a type for an enum.



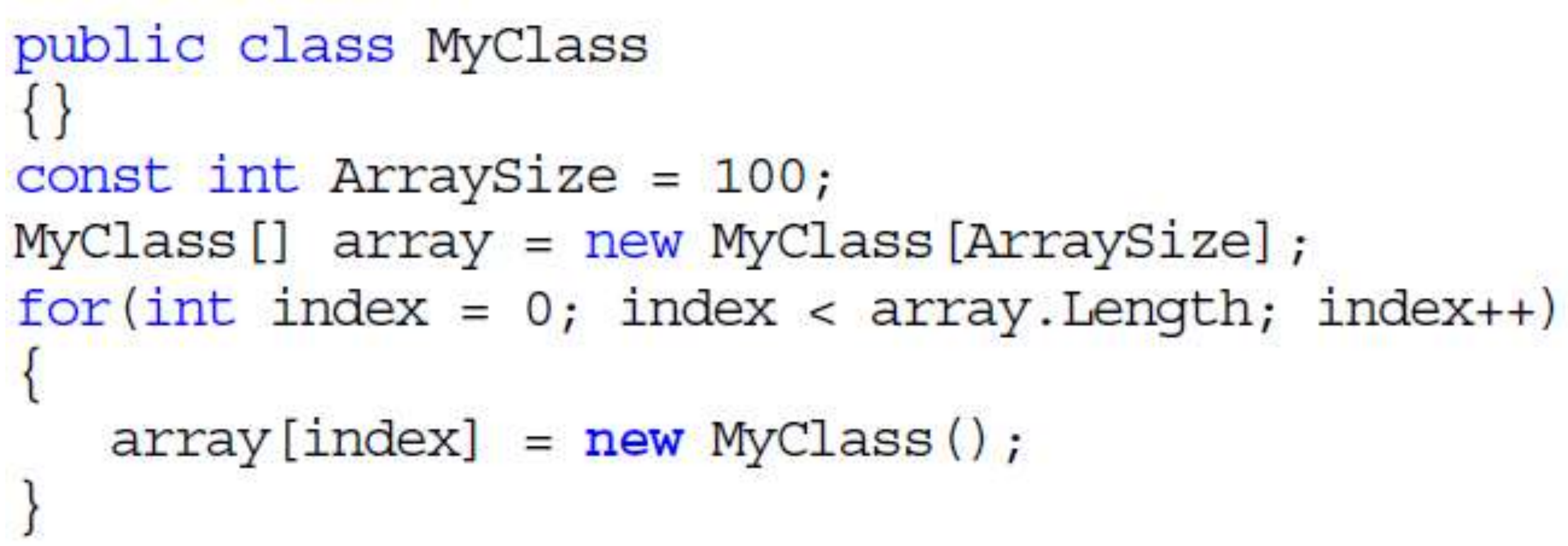
```
//Correct
public enum Color
{
    Red,Green,Blue
}
//Avoid
public enum Color
{
    Red   = 1,
    Green = 2,
    Blue  = 3
}
```



```
//Avoid
public enum Color : long
{
    Red,Green,Blue
}
```

Coding Practices

Always explicitly initialize an array of reference types using a for loop.

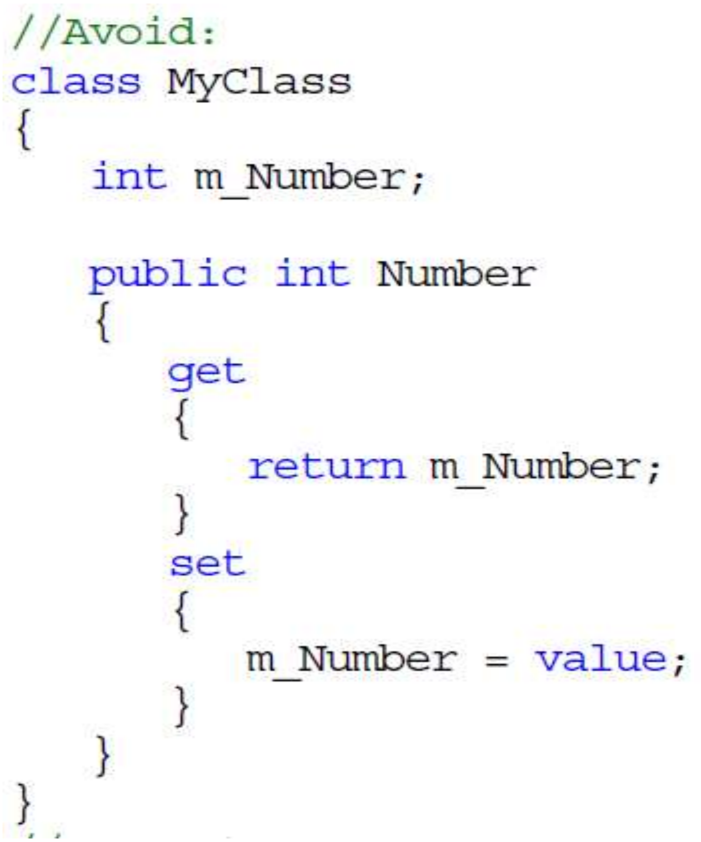


```
public class MyClass
{}
const int ArraySize = 100;
MyClass[] array = new MyClass[ArraySize];
for(int index = 0; index < array.Length; index++)
{
    array[index] = new MyClass();
}
```

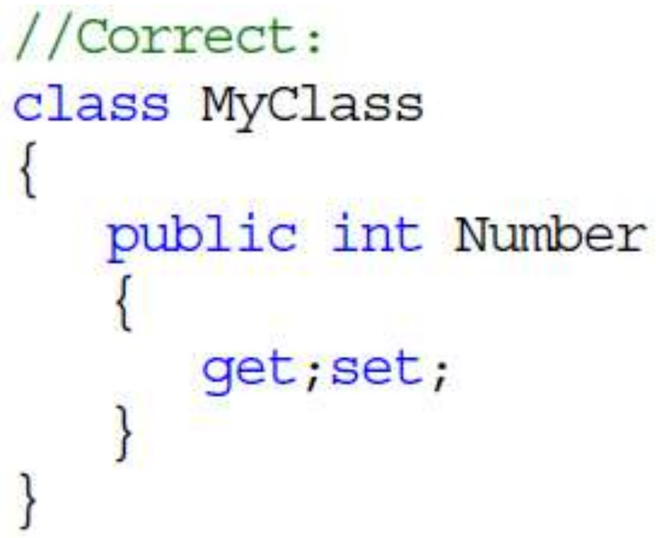
Coding Practices

Do not provide public or protected member variables. Use properties instead.

Avoid explicit properties that do nothing except access a member variable. Use automatic properties instead.



```
//Avoid:
class MyClass
{
    int m_Number;

    public int Number
    {
        get
        {
            return m_Number;
        }
        set
        {
            m_Number = value;
        }
    }
}
```



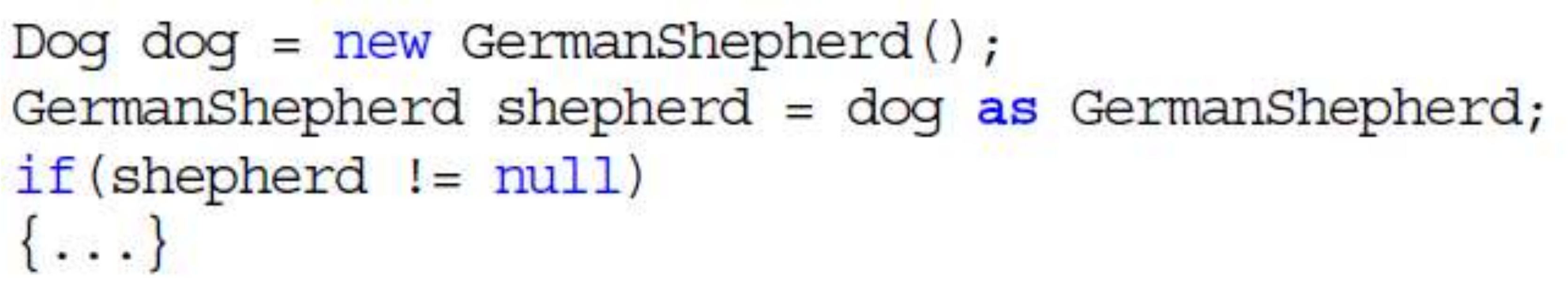
```
//Correct:
class MyClass
{
    public int Number
    {
        get;set;
    }
}
```

Coding Practices

Avoid using the new inheritance qualifier. Use override instead.

Always mark public and protected methods as virtual in a non-sealed class.

Avoid explicit casting. Use the as operator to defensively cast to a type.

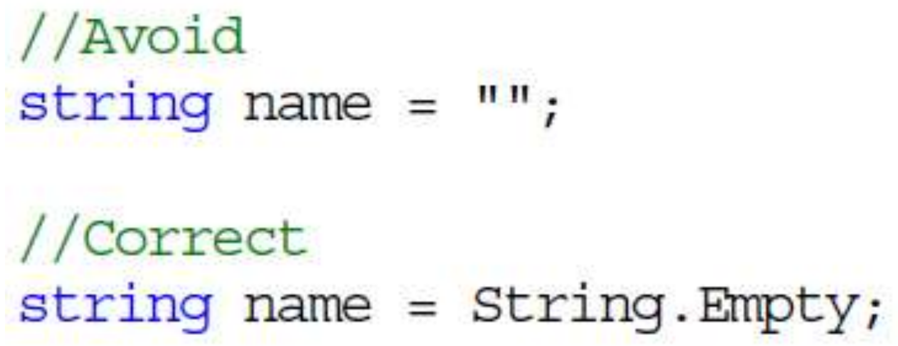


```
Dog dog = new GermanShepherd();
GermanShepherd shepherd = dog as GermanShepherd;
if(shepherd != null)
{...}
```

Coding Practices

Never hardcode strings that might change based on deployment such as connection strings.

Use String.Empty instead of "".



```
//Avoid
string name = "";

//Correct
string name = String.Empty;
```

When building a long string, use StringBuilder, not string.

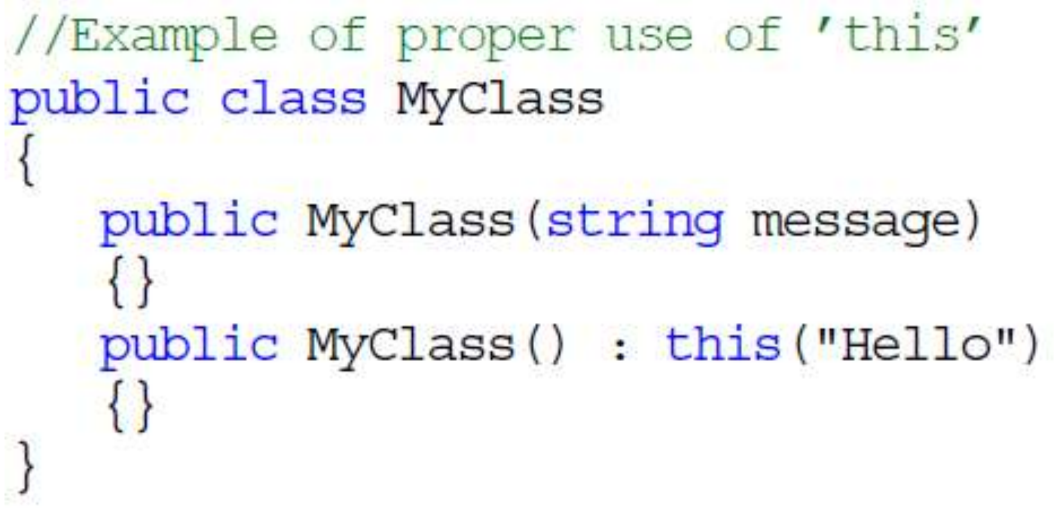
Coding Practices

Avoid providing methods on structures.

- Parameterized constructors are encouraged
- Can overload operators

Always provide a static constructor when providing static member variables.

Do not use this reference unless invoking another constructor from within a constructor



```
//Example of proper use of 'this'
public class MyClass
{
   public MyClass(string message)
   {}
   public MyClass() : this("Hello")
   {}
}
```

Coding Practices

Avoid casting to and from System.Object in code that uses generics. Use constraints or the as operator instead.

```
class SomeClass
{}
//Avoid:
class MyClass<T>
{
    void SomeMethod(T t)
    {
        object temp = t;
        SomeClass obj = (SomeClass)temp;
    }
```

```
//Correct:
class MyClass<T> where T : SomeClass
{
    void SomeMethod(T t)
    {
        SomeClass obj = t;
    }
}
```

ASP.NET Guidelines

Avoid putting code in ASPX files of ASP.NET. All code should be in the code-beside partial class.

Code in code-beside partial class of ASP.NET should call other components rather than contain direct business logic.

Always check a session variable for null before accessing it.

Avoid setting the Auto-Postback property of server controls in ASP.NET to True

Prefix user control name with “uc”.

- The rest of the user control name should be in Pascal Casing (e.g. ucMyLoginControl)

ASP.NET Guidelines

Do not store large objects in sessions, it may consume lot of server memory depending on the number of users.

Always use Style Sheets to control the look and feel of the pages.

Categories data and store it in predefined folder depending on its working. (e.g. store all images, sounds, video files in media or image folder)

Application scope

In case of multistep update where the update is treated as an atomic operation, lock application state before accessing it.

```
Application.Lock();
Application[“itemsSold”] = (int) Application[“itemsSold”] +1;
Application[“itemsLeft”] = (int) Application[“itemsLeft”] - 1;
Application.UnLock();
```

By locking the application state before updating it and unlocking it afterword's, it is ensured that another request being processed on another thread does not read application state at exactly the wrong time and see an inconsistent view of it.

Session Scope

By default the session state variables are set to true.

If this is not required in the application, these should be set to false, because maintaining session state consumes memory and processing time.

To disable session state for a page, set the EnableSessionState attribute of @Page directive to False.

```
<%@ Page EnableSessionState="False“ %>
```

If the page requires access to session variables, but will not create or modify them, set the EnableSessionState attribute of @Page directive to ReadOnly.

Session Scope

The session state may be globally disabled for the entire application, by a setting in machine.config file.

Selection of Session mode should be done judiciously.

- The default InProc mode is the fastest, but should not be used in a web-farm scenario.
- Using out-of-Proc State Server is slower than the In-Proc and has a limitation of 2 GB storage.
- The usage of SQL Server as the state data store is the slowest but has a highest scalability capability.

Data Access Guidelines using ADO.NET

Open connections late and close them early

Connection needs to be closed explicitly after usage.

- If not closed, it doesn’t get closed until the object that wraps it is garbage collected.

Store connection strings securely

Storing Connection Strings in external files like .config

When do you use DataReader?

- **Data Reader should be used in application when:**
  - There is no need to cache data
  - Result of query is too large to fit in memory
  - Data Access should be quick and in forward-only manner

- **To improve performance while using Data Reader:**
  - Close the DataReader before any of the output parameters are accessed that are associated with the command
  - Use the CommandBehavior.SequentialAccess in the ExecuteReader method to improve performance
    - The data is loaded into memory only when requested

When do you use DataSet?

- **DataSet should be used in the application when:**
  - Navigating between multiple discrete results
  - Performing data manipulation from multiple sources
  - Reusing the same set of results to perform sorting, searching, and so on.
    - Caching the results improves performance
  - Using Typed DataSet to include type checking at design time

Provider

Use SQL Data Provider and not OLEDB provider for SQL Server data access since it gives better performance

- OLEDB should be used for connecting to SQL Server 6.5 or earlier

Use Stored Procedures wherever possible, instead of firing direct queries.

WCF Guidelines

General Design Guidelines:

All services must adhere to these principles:

- Services are Secure
- Service operations leave the system in a consistent state.
- Services are thread-safe and can be accessed by concurrent clients
- Services are reliable
- Services are robust

Services can optionally adhere to these principles

- Services are interoperable
- Services are available
- Services are responsive

WCF Coding Essentials

Place Service Code in class library, not in any hosting EXE.

Do not provide a parameterized constructors to a Service class, unless it is a singleton that is hosted explicitly.

Enable reliability in the relevant bindings

Provide a meaningful namespace for contracts. For outword-facing services, use Company’s URL or equivalent URI with a year and month to support versioning.

```
[ServiceContract(namespace="http://www.IGATE.com/2014/04")]
    public interface IMyContract
     { … }
```

WCF Coding Essentials

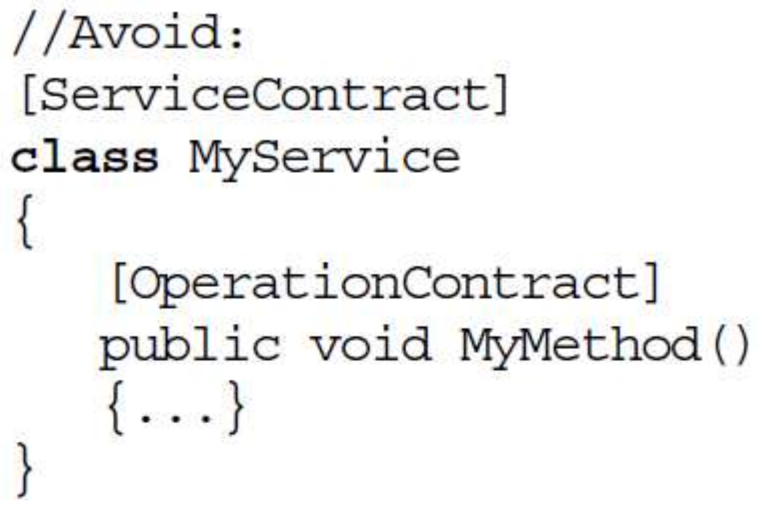
Do not mix and match named bindings with default bindings.

Do not mix and match named behaviors with default behaviors.

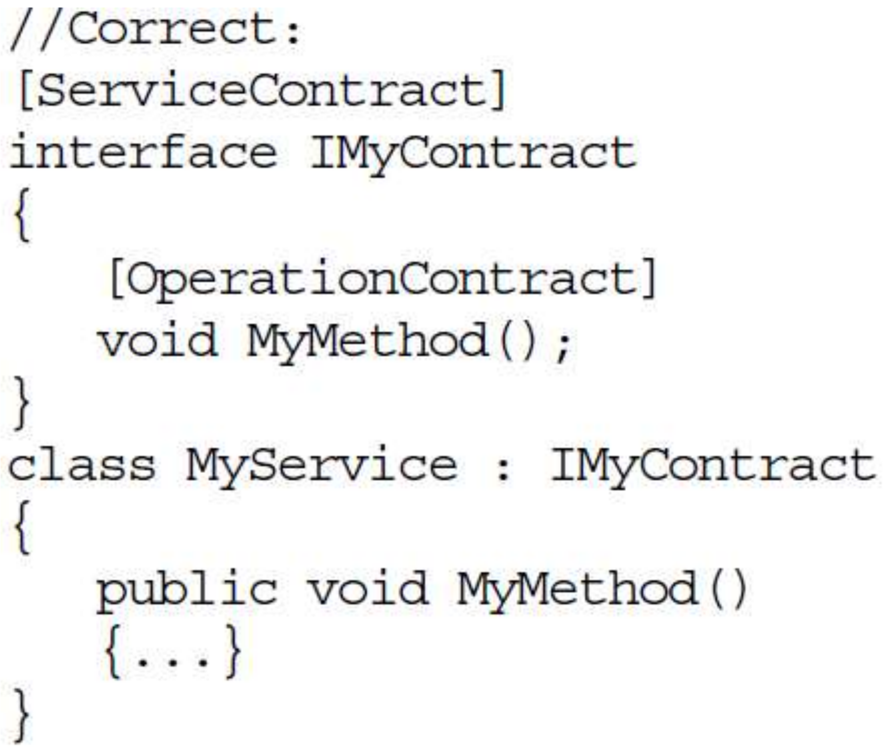
Always close or dispose of the proxy.

Service Contracts

Always apply the ServiceContract attribute on an interface, not a class.



```
//Avoid:
[ServiceContract]
class MyService
{
   [OperationContract]
   public void MyMethod()
   {...}
}
```



```
//Correct:
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
class MyService : IMyContract
{
   public void MyMethod()
   {...}
}
```

Service Contracts

Avoid Property like operations.

```
//Avoid:
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   string GetName();

   [OperationContract]
   void SetName(string name);
}
```

Service Contracts

Avoid contracts with one member.

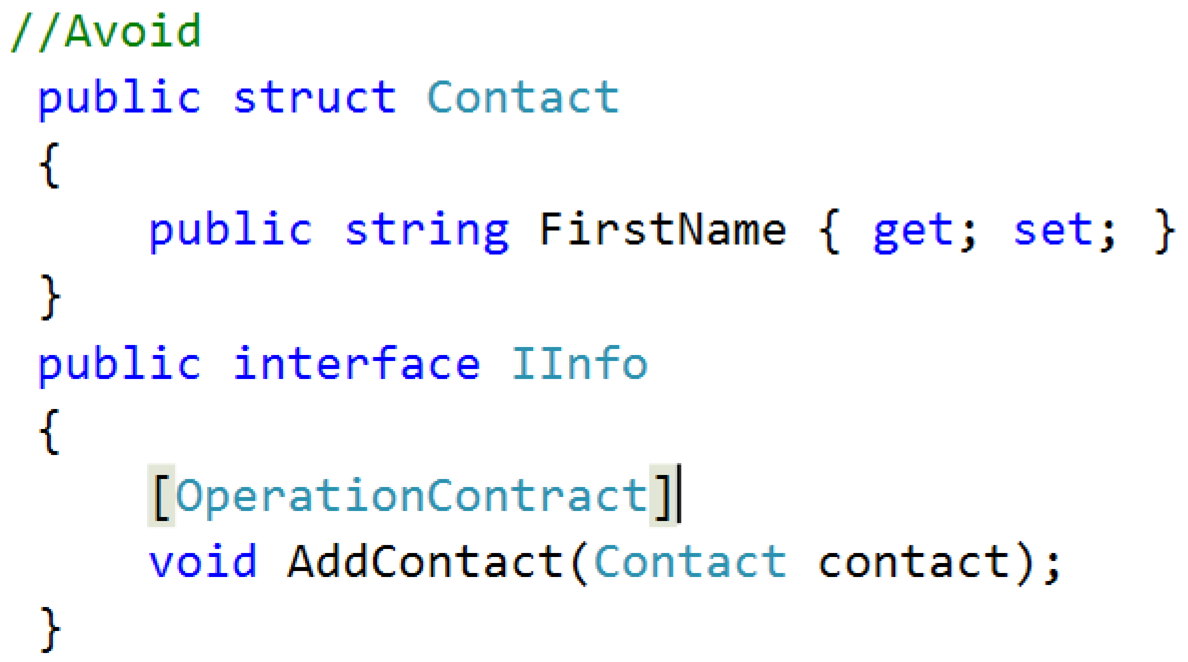
Strive to have three to five members per service contract.

Do not have more than 20 members per service contract. Twelve is the practical limit.

Data Contracts

Avoid inferred Data Contracts (POCO). Instead, be explicit and apply the DataContract attribute.

This will enable you to tap into data contract features such as versioning.



```
//Avoid
 public struct Contact
 {
     public string FirstName { get; set; }
 }
 public interface IInfo
 {
     [OperationContract]
     void AddContact(Contact contact);
 }
```

Data Contracts

Use DataMember attribute only on properties or read-only public members.

```
//Correct
[DataContract]
 public class Contact
 {
     [DataMember]
     public string FirstName { get; set; }
 }
 public interface IInfo
 {
     [OperationContract]
     void AddContact(Contact contact);
 }
```

Data Contracts

Avoid explicit XML Serialization of your own types.

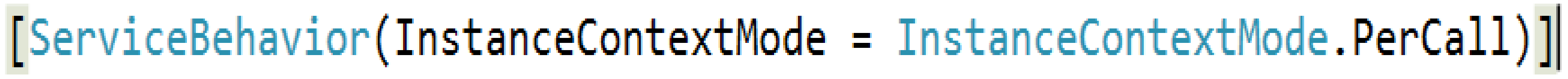
Avoid Message Contracts, as far as possible.

Do not mark delegates and events as data members.

Do not pass .NET specific types, such as Type, as operation parameters.

Instance Management

Prefer the Per-Call Instance mode when scalability is a concern.



```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
```

If setting SessionMode.NotAllowed on the contract, always configure the service instancing mode as InstanceContextMode.PerCall.

Do not mix sessionful contracts and sessionless contracts in the same service.

Operations and Calls

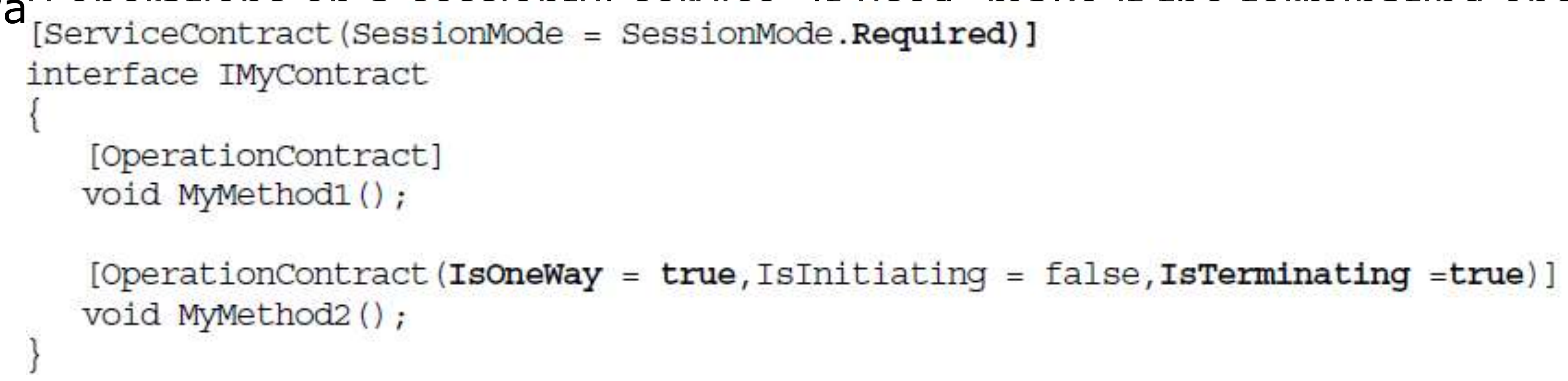
Do not treat one-way calls as asynchronous calls

Do not treat one-way calls as concurrent calls

Except Exceptions from a One-Way operation

Enable reliability even on One-Way Calls

Avoid one-way operations on a sessionful service. If used, make it the terminating operation.

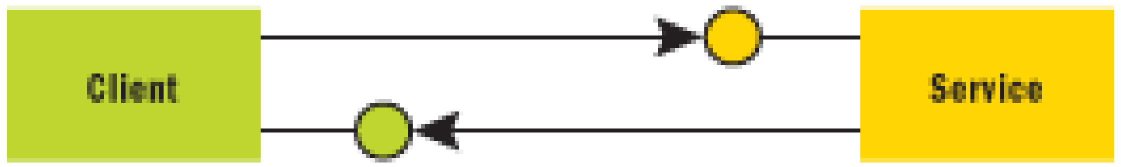


```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void MyMethod1();

   [OperationContract(IsOneWay = true,IsInitiating = false,IsTerminating =true)]
   void MyMethod2();
}
```

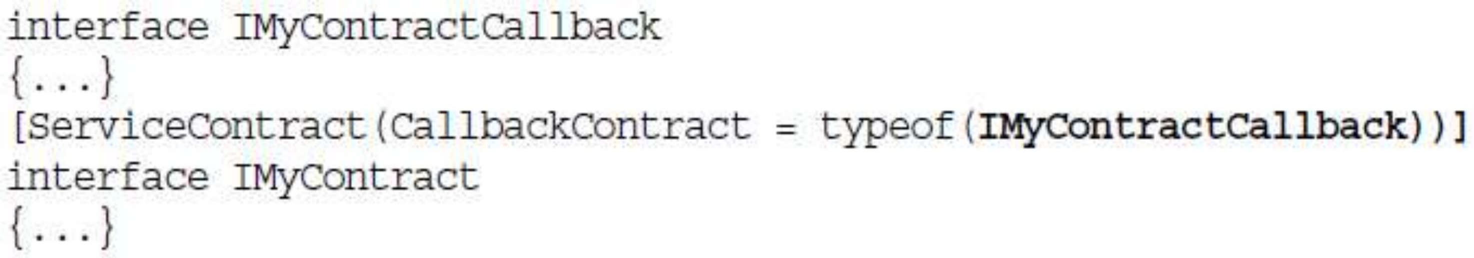
Operations and Calls

Callback operations:

- Windows Communication Foundation supports allowing the service to call back to its clients.
- During a callback, in many respects the tables are turned: the service is the client and the client becomes the service



Name the callback contract on the service side after the service contract name, suffixed by Callback



```
interface IMyContractCallback
{...}
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{...}
```

Summary

In this lesson, you have learnt:

- Good coding practices for .NET