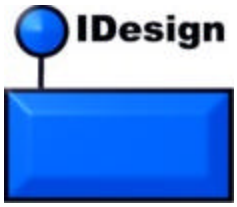


C# Coding Standard

Guidelines and Best Practices
Version 2.2

Author: Juval Lowy
www.idesign.net

Table of Content

Preface

A comprehensive coding standard is essential for a successful product delivery. The standard helps in enforcing best practices and avoiding pitfalls, and makes knowledge dissemination across the team easier. Traditionally, coding standards are thick, laborious documents, spanning hundreds of pages and detailing the rationale behind every directive. While these are still better than no standard at all, such efforts are usually indigestible by the average developer. In contrast, the C# coding standard presented here is very thin on the “why” and very detailed on the “what” and the “how.” I believe that while fully understanding every insight that goes into a particular programming decision may require reading books and even years of experience, applying the standard should not. When absorbing a new developer into your team, you should be able to simply point him or her at the standard and say: "Read this first." Being able to comply with a good standard should come before fully understanding and appreciating it—that should come over time, with experience. The coding standard presented next captures best practices, dos and don'ts, pitfalls, guidelines, and recommendations, as well as naming conventions and styles, project settings and structure, and framework-specific guidelines. Since I first published this standard for C# 1.1 in 2003, it has become the de-facto industry standard for C# and .NET development.

Juval Lowy
April 2005

1. Naming Conventions and Style

1. Use Pascal casing for type and method names and constants:

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

2. Use camel casing for local variable names and method arguments.

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

3. Prefix interface names with `I`

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

4. Prefix private member variables with `m_`. Use Pascal casing for the rest of a member variable name following the `m_`.

```
public class SomeClass
{
   private int m_Number;
}
```

5. Suffix custom attribute classes with `Attribute`.
6. Suffix custom exception classes with `Exception`.
7. Name methods using verb-object pair, such as `ShowDialog()`.
8. Methods with return values should have a name describing the value returned, such as `GetObjectState()`.
9. Use descriptive variable names.
   a) Avoid single character variable names, such as `i` or `t`. Use `index` or `temp` instead.
   b) Avoid using Hungarian notation for public or protected members.
   c) Do not abbreviate words (such as `num` instead of `number`).
10. Always use C# predefined types rather than the aliases in the `System` namespace. For example:

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

11. 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>
{...}
```

12. Use meaningful namespaces such as the product name or the company name.
13. Avoid fully qualified type names. Use the `using` statement instead.
14. Avoid putting a `using` statement inside a namespace.
15. 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;
```

16. Use delegate inference instead of explicit delegate instantiation.

```
delegate void SomeDelegate();
public void SomeMethod()
{...}
SomeDelegate someDelegate = SomeMethod;
```

17. Maintain strict indentation. Do not use tabs or nonstandard indentation, such as one space. Recommended values are three or four spaces, and the value should be uniform across.
18. Indent comments at the same level of indentation as the code you are documenting.
19. All comments should pass spell checking. Misspelled comments indicate sloppy development.
20. 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()
   {}
}
```

21. Declare a local variable as close as possible to its first use.
22. A file name should reflect the class it contains.
23. When using partial types and allocating a part per file, name each file after the logical part that part plays. For example:

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

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

25. With anonymous methods, mimic the code layout of a regular method, aligned with the anonymous delegate declaration. (complies with placing an open curly brace in a new line):

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

26. 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");
                             };
```

2 Coding Practices

1. Avoid putting multiple classes in a single file.
2. A single file should contribute types to only a single namespace. Avoid having multiple namespaces in the same file.
3. Avoid files with more than 500 lines (excluding machine-generated code).
4. Avoid methods with more than 200 lines.
5. Avoid methods with more than 5 arguments. Use structures for passing multiple arguments.
6. Lines should not exceed 120 characters.
7. Do not manually edit any machine-generated code.
   a) If modifying machine generated code, modify the format and style to match this coding standard.
   b) Use partial classes whenever possible to factor out the maintained portions.
8. Avoid comments that explain the obvious. Code should be self-explanatory. Good code with readable variable and method names should not require comments.
9. Document only operational assumptions, algorithm insights and so on.
10. Avoid method-level documentation.
    a) Use extensive external documentation for API documentation.
    b) Use method-level comments only as tool tips for other developers.
11. With the exception of zero and one, never hard-code a numeric value; always declare a constant instead.
12. Use the `const` directive only on natural constants such as the number of days of the week.
13. Avoid using `const` on read-only variables. For that, use the `readonly` directive.

```
public class MyClass
{
   public const int DaysInWeek = 7;
   public readonly int Number;
   public MyClass(int someValue)
   {
      Number = someValue;
   }
}
```

14. Assert every assumption. On average, every fifth line is an assertion.

```
using System.Diagnostics;

object GetObject()
{...}

object someObject = GetObject();
Debug.Assert(someObject != null);
```

15. Every line of code should be walked through in a “white box” testing manner.
16. Catch only exceptions for which you have explicit handling.

17. In a `catch` statement that throws an exception, always throw the original exception (or another exception constructed from the original exception) to maintain the stack location of the original error:

```
catch(Exception exception)
{
   MessageBox.Show(exception.Message);
   throw exception;
}
```

18. Avoid error codes as method return values.
19. Avoid defining custom exception classes.
20. When defining custom exceptions:
    a) Derive the custom exception from `Exception`.
    b) Provide custom serialization.
21. Avoid multiple `Main()` methods in a single assembly.
22. Make only the most necessary types public, mark others as `internal`.
23. Avoid friend assemblies, as they increase inter-assembly coupling.
24. Avoid code that relies on an assembly running from a particular location.
25. Minimize code in application assemblies (EXE client assemblies). Use class libraries instead to contain business logic.
26. Avoid providing explicit values for enums unless they are integer powers of 2:

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

27. Avoid specifying a type for an enum.

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

28. Always use a curly brace scope in an `if` statement, even if it conditions a single statement.
29. Avoid using the ternary conditional operator.

30. Avoid function calls in Boolean conditional statements. Assign into local variables and check on them.

```
bool IsEverythingOK()
{...}
//Avoid:
if(IsEverythingOK())
{...}
//Correct:
bool ok = IsEverythingOK();
if(ok)
{...}
```

31. Always use zero-based arrays.
32. With indexed collection, use zero-based indexes
33. 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();
}
```

34. Do not provide public or protected member variables. Use properties instead.
35. Avoid using the `new` inheritance qualifier. Use `override` instead.
36. Always mark public and protected methods as `virtual` in a non-sealed class.
37. Never use unsafe code, except when using interop.
38. 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)
{...}
```

39. Always check a delegate for `null` before invoking it.

40. Do not provide public event member variables. Use event accessors instead.

```
public class MyPublisher
{
   MyDelegate m_SomeEvent;
   public event MyDelegate SomeEvent
   {
      add
      {
         m_SomeEvent += value;
      }
      remove
      {
         m_SomeEvent -= value;
      }
   }
}
```

41. Avoid defining event-handling delegates. Use `EventHandler<T>` or `GenericEventHandler` instead. `GenericEventHandler` is defined in Chapter 6 of Programming .NET Components 2nd Edition.
42. Avoid raising events explicitly. Use `EventsHelper` to publish events defensively. `EventsHelper` is presented in Chapter 6-8 of Programming .NET Components 2nd Edition.
43. Always use interfaces. See Chapters 1 and 3 in Programming .NET Components 2nd Edition.
44. Classes and interfaces should have at least 2:1 ratio of methods to properties.
45. Avoid interfaces with one member.
46. Strive to have three to five members per interface.
47. Do not have more than 20 members per interface. Twelve is probably the practical limit.
48. Avoid events as interface members.
49. When using abstract classes, offer an interface as well.
50. Expose interfaces on class hierarchies.
51. Prefer using explicit interface implementation.
52. Never assume a type supports an interface. Defensively query for that interface.

```
SomeType obj1;
IMyInterface obj2;

/* Some code to initialize obj1, then: */
obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
   obj2.Method1();
}
else
{
   //Handle error in expected interface
}
```

53. Never hardcode strings that will be presented to end users. Use resources instead.
54. Never hardcode strings that might change based on deployment such as connection strings.
55. Use `String.Empty` instead of `""`:

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

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

56. When building a long string, use `StringBuilder`, not `string`.
57. Avoid providing methods on structures.
    a) Parameterized constructors are encouraged.
    b) Can overload operators.
58. Always provide a static constructor when providing static member variables.
59. Do not use late-binding invocation when early-binding is possible.
60. Use application logging and tracing.
61. Never use `goto` unless in a `switch` statement fall-through.
62. Always have a `default` case in a `switch` statement that asserts.

```
int number = SomeMethod();
switch(number)
{
   case 1:
      Trace.WriteLine("Case 1:");
      break;
   case 2:
      Trace.WriteLine("Case 2:");
      break;
   default:
      Debug.Assert(false);
      break;
}
```

63. Do not use the `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")
   {}
}
```

64. Do not use the `base` word to access base class members unless you wish to resolve a conflict with a subclasses member of the same name or when invoking a base class constructor.

```
//Example of proper use of 'base'
public class Dog
{
   public Dog(string name)
   {}
   virtual public void Bark(int howLong)
   {}
}
public class GermanShepherd : Dog
{
   public GermanShepherd(string name): base(name)
   {}
   override public void Bark(int howLong)
   {
      base.Bark(howLong);
   }
}
```

65. Do not use `GC.AddMemoryPressure()`.

66. Do not rely on `HandleCollector`.

67. Implement `Dispose()` and `Finalize()` methods based on the template in Chapter 4 of Programming .NET Components 2nd Edition.

68. Always run code unchecked by default (for the sake of performance), but explicitly in checked mode for overflow- or underflow-prone operations:

```
int CalcPower(int number,int power)
{
   int result = 1;
   for(int count = 1;count <= power;count++)
   {
      checked
      {
         result *= number;
      }
   }
   return result;
}
```

69. Avoid explicit code exclusion of method calls (`#if…#endif`). Use conditional methods instead:

```
public class MyClass
{
   [Conditional("MySpecialCondition")]
   public void MyMethod()
   {}
}
```

70. 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;
   }
}
```

71. Do not define constraints in generic interfaces. Interface-level constraints can often be replaced by strong-typing.

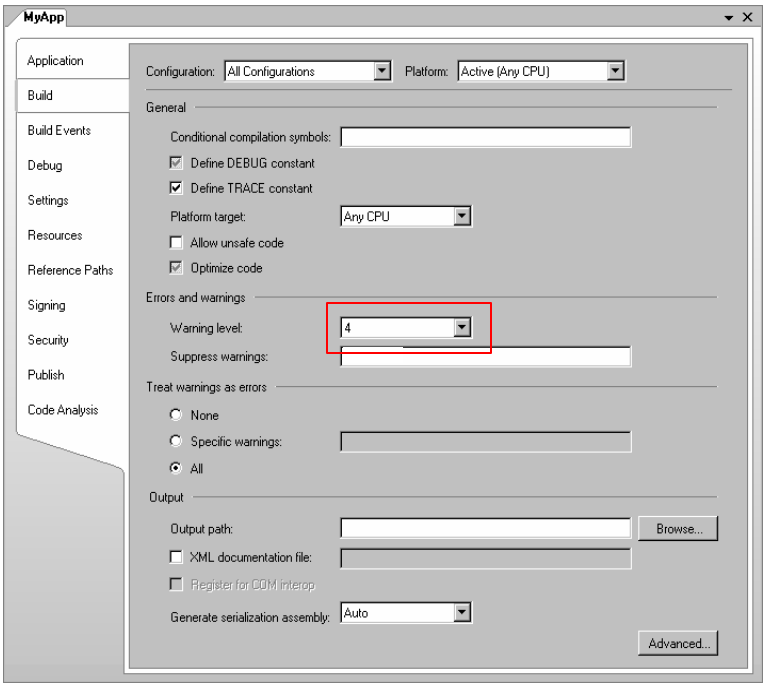
```
public class Customer
{...}
//Avoid:
public interface IList<T> where T : Customer
{...}
//Correct:
public interface ICustomerList : IList<Customer>
{...}
```

72. Do not define method-specific constraints in interfaces.
73. Do not define constraints in delegates.
74. If a class or a method offers both generic and non generic flavors, always prefer using the generics flavor.
75. When implementing a generic interface that derives from an equivalent non-generic interface (such as `IEnumerable<T>`), use explicit interface implementation on all methods, and implement the non-generic methods by delegating to the generic ones:

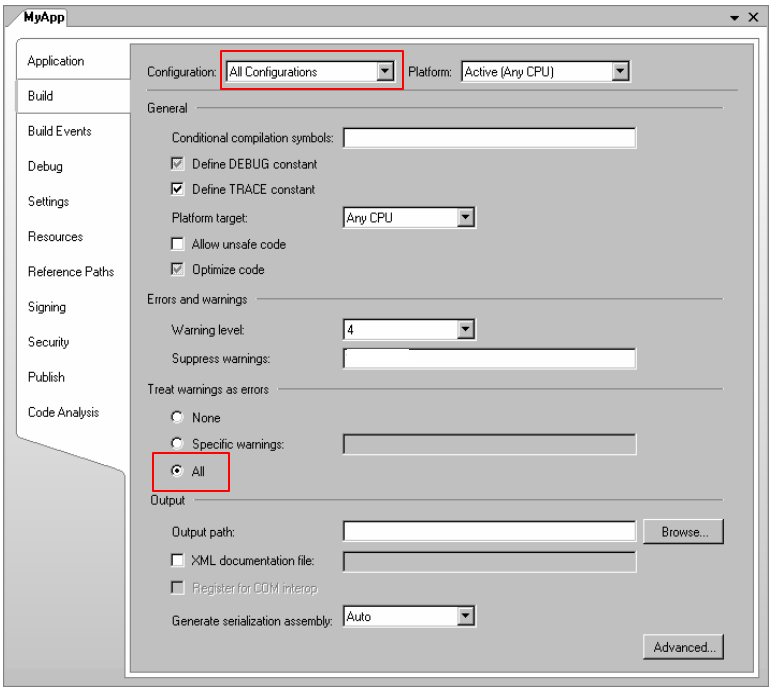
```
class MyCollection<T> : IEnumerable<T>
{
   IEnumerator<T> IEnumerable<T>.GetEnumerator()
   {...}
   IEnumerator IEnumerable.GetEnumerator()
   {
      IEnumerable<T> enumerable = this;
      return enumerable.GetEnumerator();
   }
}
```

3 Project Settings and Project Structure

1. Always build your project with warning level 4



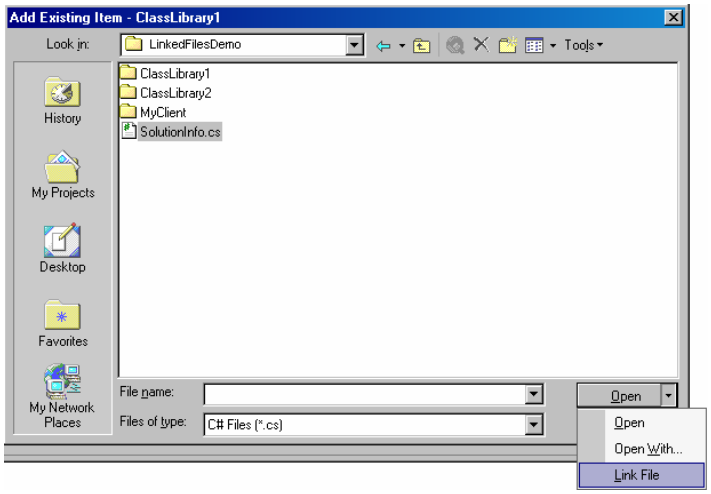
2. Treat warnings as errors in the Release build (note that this is not the default of Visual Studio). Although it is optional, this standard recommends treating warnings as errors in Debug builds as well.

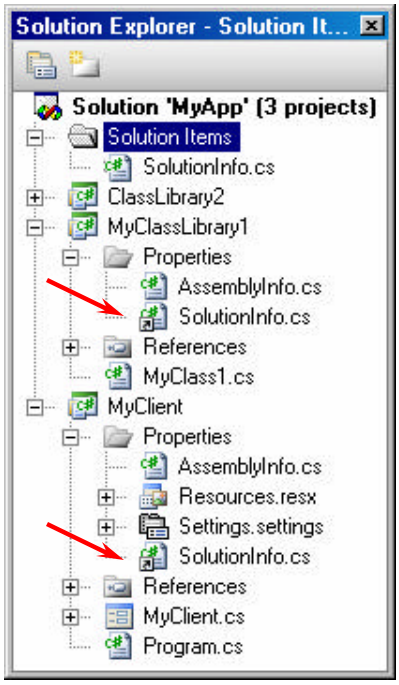


3. Avoid suppressing specific compiler warnings.
4. Always explicitly state your supported runtime versions in the application configuration file.

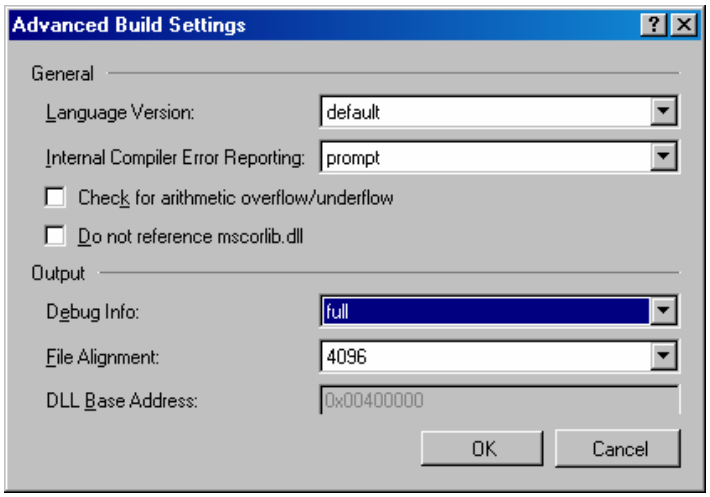
```
<?xml version="1.0"?>
<configuration>
   <startup>
      <supportedRuntime version="v2.0.50727.0"/>
      <supportedRuntime version="v1.1.4322.0"/>
   </startup>
</configuration>
```

5. Avoid explicit custom version redirection and binding to CLR assemblies.
6. Avoid explicit preprocessor definitions (`#define`). Use the project settings for defining conditional compilation constants.
7. Do not put any logic inside *AssemblyInfo.cs*.
8. Do not put any assembly attributes in any file besides *AssemblyInfo.cs*.
9. Populate all fields in *AssemblyInfo.cs* such as company name, description, and copyright notice.
10. All assembly references in the same solution should use relative paths.
11. Disallow cyclic references between assemblies.
12. Avoid multi-module assemblies.
13. Avoid tampering with exception handling using the Exception window (Debug|Exceptions).
14. Strive to use uniform version numbers on all assemblies and clients in the same logical application (typically a solution). Use the *SolutionInfo.cs* technique from Chapter 5 of Programming .NET Components 2nd Edition to automate.

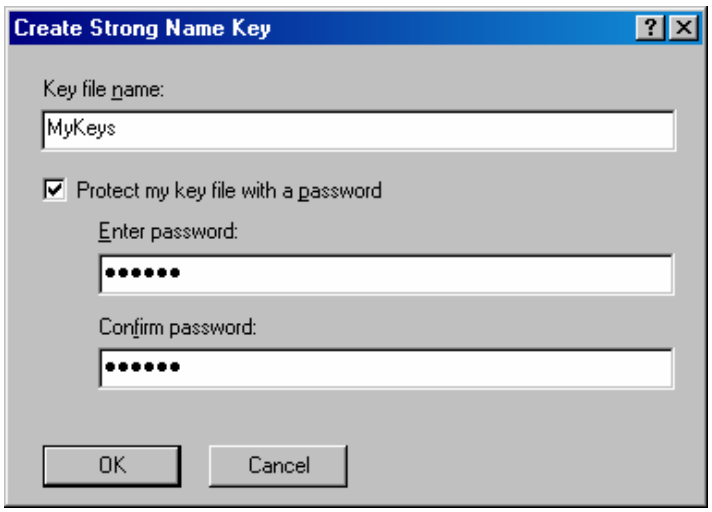




15. Link all solution-wide information to a global shared *SolutionInfo.cs* file.
16. Name your application configuration file as *App.config*, and include it in the project.
17. Modify Visual Studio 2005 default project structure to comply with your project standard layout, and apply uniform structure for project folders and files.
18. A Release build should contain debug symbols.



19. Always sign your assemblies, including the client applications.
20. Use password-protected keys.



4 Framework Specific Guidelines

4.1 Data Access

1. Always use type-safe data sets or data tables. Avoid raw ADO.NET.
2. Always use transactions when accessing a database.
   a) Always use Enterprise Services or System.Transactions transactions.
   b) Do not use ADO.NET transactions by enlisting the database explicitly.
3. Always use transaction isolation level set to Serializable. Management decision is required to use anything else.
4. Do not use the Data Source window to drop connections on windows forms, ASP.NET forms or web services. Doing so couples the presentation tier to the data tier.
5. Avoid SQL Server authentication. Use Windows authentication instead.
6. Run components accessing SQL Server under separate identity from that of the calling client.
7. Always wrap your stored procedures in a high level, type safe class. Only that class invokes the stored procedures. Let Visual Studio 2005 type-safe data adapters automate as much of that as possible.
8. Avoid putting any logic inside a stored procedure. If you have anything more complex than simple switching logic to vary your query based on the parameter values, you should consider putting that logic in the business logic of the consuming code.

4.2 ASP.NET and Web Services

1. Avoid putting code in ASPX files of ASP.NET. All code should be in the code-beside partial class.
2. Code in code beside partial class of ASP.NET should call other components rather than contain direct business logic.
3. Always check a session variable for `null` before accessing it.
4. In transactional pages or web services, always store session in SQL server.
5. Avoid setting the Auto-Postback property of server controls in ASP.NET to True.
6. Turn on Smart Navigation for ASP.NET pages.
7. Strive to provide interfaces for web services. See Appendix A of Programming .NET Components 2nd Edition.
8. Always provide a namespace and service description for web services.
9. Always provide a description for web methods.
10. When adding a web service reference, provide a meaningful name for the location.

11. In both ASP.NET pages and web services, wrap a session variable in a local property. Only that property is allowed to access the session variable, and the rest of the code uses the property, not the session variable.

```
public class Calculator : WebService
{
   int Memory
   {
      get
      {
         int memory = 0;
         object state = Session["Memory"];
         if(state != null)
         {
            memory = (int)state;
         }
         return memory;
      }
      set
      {
         Session["Memory"] = value;
      }
   }
   [WebMethod(EnableSession=true)]
   public void MemoryReset()
   {
      Memory = 0;
   }
}
```

12. Always modify a client-side web service wrapper class to support cookies, since you have no way of knowing whether the service uses Session state or not.

```
public class Calculator : SoapHttpClientProtocol
{
   public Calculator()
   {
      CookieContainer = new System.Net.CookieContainer();
      Url = ...;
   }
}
```

4.3 Multithreading

1. Use Synchronization Domains. See Chapter 8 in Programming .NET Components 2nd Edition. Avoid manual synchronization because that often leads to deadlocks and race conditions.
2. Never call outside your synchronization domain.
3. Manage asynchronous call completion on a callback method. Do not wait, poll, or block for completion.
4. Always name your threads. The name is traced in the debugger Threads window, making debug sessions more productive.

```
Thread currentThread = Thread.CurrentThread;
string threadName = "Main UI Thread";
currentThread.Name = threadName;
```

5. Do not call `Suspend()` or `Resume()` on a thread.
6. Do not call `Thread.Sleep()`, except in the following conditions:
   a) `Thread.Sleep(0)` is an acceptable optimization technique to force a context switch.
   b) `Thread.Sleep()` is acceptable in testing or simulation code.
7. Do not call `Thread.SpinWait()`.
8. Do not call `Thread.Abort()` to terminate threads. Use a synchronization object instead to signal the thread to terminate. See Chapter 8 in Programming .NET Components 2nd Edition.
9. Avoid explicitly setting thread priority to control execution. You can set thread priority based on task semantic, such as below normal (`ThreadPriority.BelowNormal`) for a screen saver.
10. Do not read the value of the `ThreadState` property. Use `Thread.IsAlive` to determine whether the thread is dead or alive.
11. Do not rely on setting the thread type to background thread for application shutdown. Use a watchdog or other monitoring entity to deterministically kill threads.
12. Do not use thread local storage unless thread affinity is guaranteed.
13. Do not call `Thread.MemoryBarrier()`.
14. Never call `Thread.Join()` without checking that you are not joining your own thread.

```
void WaitForThreadToDie(Thread thread)
{
   Debug.Assert(Thread.CurrentThread.ManagedThreadId != thread.ManagedThreadId);
   thread.Join();
}
```

15. Always use the `lock()` statement rather than explicit `Monitor` manipulation.
16. Always encapsulate the `lock()` statement inside the object it protects.

```
public class MyClass
{
   public void DoSomething()
   {
      lock(this)
      {...}
   }
}
```

17. You can use synchronized methods instead of writing the `lock()` statement yourself.
18. Avoid fragmented locking (see Chapter 8 of Programming .NET Components 2nd Edition).
19. Avoid using a `Monitor` to wait or pulse objects. Use manual or auto-reset events instead.
20. Do not use volatile variables. Lock your object or fields instead to guarantee deterministic and thread-safe access. Do not use `Thread.VolatileRead()`, `Thread.VolatileWrite()`, or the `volatile` modifier.

21. Avoid increasing the maximum number of threads in the thread pool.
22. Never stack `lock` statements because that does not provide atomic locking. Use `WaitHandle.WaitAll()` instead.

```
MyClass obj1 = new MyClass();
MyClass obj2 = new MyClass();
MyClass obj3 = new MyClass();

//Do not stack lock statements
lock(obj1)
lock(obj2)
lock(obj3)
{
   obj1.DoSomething();
   obj2.DoSomething();
   obj3.DoSomething();
}
```

4.4 Serialization

1. Prefer the binary formatter.
2. Mark serialization event handling methods as private.
3. Use the generic `IGenericFormatter` interface. See Chapter 9 of Programming .NET Components 2nd Edition.
4. Mark non-sealed classes as serializable.
5. When implementing `IDeserializationCallback` on a non-sealed class, make sure to do so in a way that allowed subclasses to call the base class implementation of `OnDeserialization()`.See Chapter 3 of Programming .NET Components 2nd Edition.
6. Always mark un-serializable member variables as non serializable.
7. Always mark delegates on a serialized class as non-serializable fields:

```
[Serializable]
public class MyClass
{
   [field:NonSerialized]
   public event EventHandler MyEvent;
}
```

4.5 Remoting

1. Prefer administrative configuration to programmatic configuration.
2. Always implement `IDisposable` on single call objects.
3. Always prefer a TCP channel and a binary format when using remoting, unless a firewall is present.
4. Always provide a `null` lease for a singleton object.

```
public class MySingleton : MarshalByRefObject
{
   public override object InitializeLifetimeService()
   {
      return null;
   }
}
```

5. Always provide a sponsor for a client activated object. The sponsor should return the initial lease time.
6. Always unregister the sponsor on client application shutdown.
7. Always put remote objects in class libraries.
8. Avoid using SoapSuds.
9. Avoid hosting in IIS.
10. Avoid using uni-directional channels.
11. Always load a remoting configuration file in `Main()` even if the file is empty, and the application does not use remoting.

```
static void Main()
{
   RemotingConfiguration.Configure("MyApp.exe.config");
   /* Rest of Main()  */
}
```

12. Avoid using `Activator.GetObject()` and `Activator.CreateInstance()` for remote objects activation. Use `new` instead.
13. Always register port 0 on the client side, to allow callbacks.
14. Always elevate type filtering to full on both client and host to allow callbacks.

4.6 Security

1. Always demand your own strong name on assemblies and components that are private to the application, but are public (so that only you can use them).

```
public class PublicKeys
{
   public const string MyCompany = "12345678948000009400000006020000000240000"+
                                    "52534131000400000100010007D1FA57C4AED9F0"+
                                    "A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C83"+
                                    "4C99921EB23BE79AD9D5DCC1DD9AD23613210290"+
                                    "0B723CF980957FC4E177108FC607774F29E8320E"+
                                    "92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99"+
                                    "285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF"+
                                    "0FC4963D261C8A12436518206DC093344D5AD293";
}
[StrongNameIdentityPermission(SecurityAction.LinkDemand,
                              PublicKey = PublicKeys.MyCompany)]
public class MyClass
{...}
```

2. Apply encryption and security protection on application configuration files.
3. When importing an interop method, assert unmanaged code permission, and demand appropriate permission instead.

```
[DllImport("user32",EntryPoint="MessageBoxA")]
private static extern int Show(IntPtr handle,string text,string caption,
                                                                int msgType);
[SecurityPermission(SecurityAction.Assert,UnmanagedCode = true)]
[UIPermission(SecurityAction.Demand,
                                   Window = UIPermissionWindow.SafeTopLevelWindows)]
```

```
public static void Show(string text,string caption)
{
   Show(IntPtr.Zero,text,caption,0);
}
```

4. Do not suppress unmanaged code access via the `SuppressUnmanagedCodeSecurity` attribute.
5. Do not use the `/unsafe` switch of *TlbImp.exe*. Wrap the RCW in managed code so that you could assert and demand permissions declaratively on the wrapper.
6. On server machines, deploy a code access security policy that grants only Microsoft, ECMA, and self (identified by a strong name) full trust. Code originating from anywhere else is implicitly granted nothing.
7. On client machines, deploy a security policy which grants client application only the permissions to execute, to call back the server and to potentially display user interface. When not using ClickOnce, client application should be identified by a strong name in the code groups.
8. To counter a luring attack, always refuse at the assembly level all permissions not required to perform the task at hand.

```
[assembly:UIPermission(SecurityAction.RequestRefuse,
                       Window=UIPermissionWindow.AllWindows)]
```

9. Always set the principal policy in every `Main()` method to Windows

```
public class MyClass
{
   static void Main()
   {
      AppDomain currentDomain = Thread.GetDomain();
      currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
   }
   //other methods
}
```

10. Never assert a permission without demanding a different permission in its place. See Chapter 12 in Programming .NET Components 2nd Edition.

4.7 System.Transactions

1. Always dispose of a `TransactionScope` object.
2. Inside a transaction scope, do not put any code after the call to `Complete()`.
3. When setting the ambient transaction, always save the old ambient transaction and restore it when you are done.
4. In Release builds, never set the transaction timeout to zero (infinite timeout).
5. When cloning a transaction, always use `DependentCloneOption.BlockCommitUntilComplete`.
6. Create a new dependent clone for each worker thread. Never pass the same dependent clone to multiple threads.
7. Do not pass a transaction clone to the `TransactionScope`'s constructor.
8. Always catch and discard exceptions thrown by a transaction scope that is set to `TransactionScopeOption.Suppress`.

4.8 Enterprise Services

1. Do not catch exceptions in a transactional method. Use the `AutoComplete` attribute. See Chapter 4 in COM and .NET Component Services.
2. Do not call `SetComplete()`, `SetAbort()`, and the like. Use the `AutoComplete` attribute.

```
[Transaction]
public class MyComponent : ServicedComponent
{
   [AutoComplete]
   public void MyMethod(long objectIdentifier)
   {...}
}
```

3. Always override `CanBePooled` and return `true` (unless you have a good reason not to return to pool)

```
public class MyComponent : ServicedComponent
{
   protected override bool CanBePooled()
   {
      return true;
   }
}
```

4. Always call `Dispose()` explicitly on a pooled objects unless the component is configured to use JITA as well.
5. Never call `Dispose()` when the component uses JITA.
6. Always set authorization level to application and component.
7. Set authentication level to `privacy` on all applications.

```
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(
           true, //Authorization
           AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
           Authentication=AuthenticationOption.Privacy,
           ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

8. Set impersonation level on client assemblies to `Identity`.
9. Always set `ComponentAccessControl` attribute on serviced components to `true` (the default is `true`)

```
[ComponentAccessControl]
public class MyComponent : ServicedComponent
{...}
```

10. Always add to the `Marshaler` role the Everyone user

```
[assembly: SecurityRole("Marshaler",SetEveryoneAccess = true)]
```

11. Apply `SecureMethod` attribute to all classes requiring authentication.

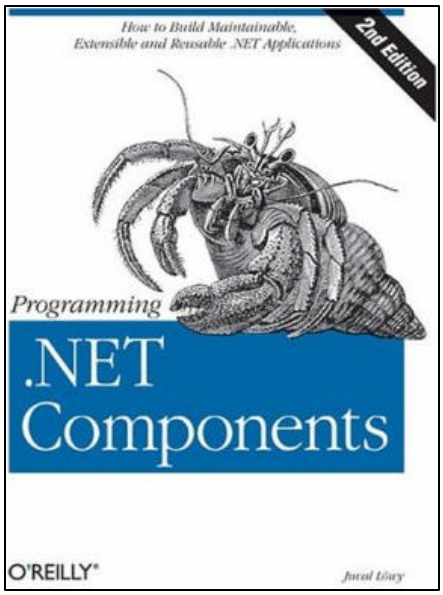
```
[SecureMethod]
public class MyComponent : ServicedComponent
{...}
```

5 Resources

5.1 Programming .NET Components 2nd Edition

By Juval Lowy, O'Reilly 2005

ISBN: 0-596-10207-0



5.2 The Advanced .NET 2.0 Master Class

This intense class is world-acclaimed due to its selection of topics, the depth of the discussion, and its focus on best practices, design guidelines, original tools and utilities, pitfalls, tips and tricks.

More at *www.idesign.net*

5.3 The IDesign Serviceware Downloads

IDesign serviceware download is a set of original techniques, tools utilities and even breakthroughs developed by the IDesign architects. The utilities are largely productivity-enhancing tools, or they compensate for some oversight in the design of .NET or WCF. The demos are also used during our *Master Classes* to demystify technical points, as lab exercises or to answer questions. The classes' attendees find the demos useful not only in class but after it. The demos serve as a starting point for new projects and as a rich reference and samples source.