C# Coding Standard

Guidelines and Best Practices
Version 3.01

Author: Juval Löwy

www.idesign.net

Table of Contents

Preface

In life it is always better to be proactive than reactive. As the saying goes, an ounce of prevention is worth a pound of medicine. Nowhere is this truer than in software development, which is one of the most intricate and challenging tasks. A comprehensive coding standard is a first-class preventive measure because complying with the standard avoids known and recurring problems, as well as enforcing best practices. This makes a good coding standard essential for successful product delivery. The standard also makes knowledge dissemination across the team easier. Aligning the code with the standard makes even a novice program like a veteran.

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 IDesign C# coding standard 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 the developer at the standard and say: "Read this first." Complying with a good standard should come before fully understanding and appreciating it—that should come over time, with experience. The coding standard in this document captures best practices, dos and don'ts, pitfalls, guidelines, and recommendations, as well as naming conventions and styles, project settings and structure, and generic programming 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. I am pleased to release here the updated version for C# 10, which also accounts for a new generation of tools for today’s distributed world.

Juval Löwy

March 2023

1 Conventions and Style

1.1 Naming

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

```
class SomeClass
{
   const int DefaultSize = 123;
   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** such as:

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

4) Prefix private member variables with **m_** or **_**. Use Pascal casing for the rest of a member variable name following the **m_**:

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

5) Suffix custom attribute classes with **Attribute**.

6) Suffix custom exception classes with **Exception**.

7) Name methods using a verb-object pair, such as **ShowDialog()**.

   a) In an interface that declares the object, name methods using only the verb.

   ```
   interface IMyDialog
   {
      void Show();
   }
   ```

8) Methods with return values should have a name describing the value returned, such as **GetLogbookState()**.

9) Use descriptive variable names.
   a) Never use single character variable names, such as **`i`** or **`t`**. Use **`index`** or **`temp`** instead.
   b) Do not abbreviate words (such as **`num`** instead of **`number`**).
10) Name a source file after the type it contains.
11) Place each part of a partial type in a separate file. Name the file after the logical part and include the type name and the part name in the file name. For example:

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

12) Use capital letters for generic type parameters. Reserve suffixing **`Type`** when dealing with the .NET type **`Type`**:

```
//Avoid:
public class Dictionary<KeyType,DataType>
{...}
public class Dictionary<TKey,TData>
{...}

//Correct:
public class Dictionary<K,T>
{...}
public class Dictionary<K,D>
{...}
```

13) Provide different color for both numbers and string literals. This standard recommends purple for numbers and teal for literals:

```
//Avoid
int number = 123;
string name = "Juval";

//Correct:
int number = 123;
string name = "Juval";
```

1.2 Namespaces

1) Use meaningful namespaces such as product name or company name.
2) Avoid fully qualified type names. Use the **`using`** statement instead.
3) Never put **`using`** statements inside a namespace.
4) Avoid multiple namespaces in the same file.
5) Group all framework namespaces together and put custom or third-party namespaces underneath, followed by solution namespaces. Order alphabetically the namespaces in a group:

```
using System;
using System.Collections.Generic;
using SomeFramework;
using MyCompany;
using MyControls;
```

1.3 Layout and Structure

1) Group relevant member variables together.
2) Group relevant methods together.
3) Group together all method implementations for the same interface.
4) Order elements of a class as follows: members, properties, constructors, methods. Have one line separation between groups.
5) Maintain strict indentation. Do not use tabs or nonstandard indentation, such as one space. This standard recommends indentation increments of three or four spaces.
6) Indent comments at the same level of indentation as the code you are documenting.
7) Declare a local variable as close as possible to its first use.
8) Always place an open curly brace (**{**) on a new line.
9) Always use curly braces scope in an **`if`** statement, even when it conditions a single statement.

```
//Never:
if(...)
   Trace.WriteLine("Avoid conditional code without scope");

//Correct:
if(...)
{
   Trace.WriteLine("Always have a scope");
}
```

10) Always use curly braces scope in a loop statement (**for**, **while**, **do**, **foreach**), even when the loop contains a single statement.

```
//Avoid
foreach(...)
   Trace.WriteLine("Avoid loops without scope");

//Correct:
foreach(...)
{
   Trace.WriteLine("Always have a scope");
}
```

11) With automatic properties, place the **get** and the **set** under the property name, this way:

```
int MyProperty
{get;set;}
```

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

```
delegate void SomeDelegate(string someString);

//Avoid
void InvokeMethod()
{
   SomeDelegate someDelegate = delegate(string name){ Trace.WriteLine(name);};
   someDelegate("Juval");
}
//Correct:
void InvokeMethod()
{
   SomeDelegate someDelegate = delegate(string name)
                               {
                                  Trace.WriteLine(name);
                               };
   someDelegate("Juval");
}
```

13) Use empty parentheses on parameter-less anonymous methods. Omit the parentheses only if you can use the anonymous method on any delegate:

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

14) With Lambda expressions, mimic the code layout of a regular method, aligned with the delegate declaration. Do not omit the variable and use parentheses:

```
delegate void SomeDelegate(string someString);

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

15) Always use curly braces on cases and **`default`** within **`switch`** statement. Align the curly braces under the **`case`** and **`default`**:

```
int number = ...;

switch(number)
{
   case 1:
   {
      ...
      break;
   }
   case 2:
   {
      ...
      break;
   }
   default:
   {
      ...
   }
}
```

1.4 Visibility

1) Do not add private visibility modifier to private members or methods:

```
//Avoid:
class MyClass
{
   private int m_MyNumber;

   private void SomeMyMethod()
   {...}
}

//Correct:
class MyClass
{
   int m_MyNumber;

   void SomeMyMethod()
   {...}
}
```

2) Do not add **`internal`** visibility modifier to internal types:

```
//Avoid:
internal class MyClass
{...}

//Correct:
class MyClass
{...}
```

3) Make only the most necessary types public. Default others to internal.

1.5 Files and Lines

1) Avoid files with more than 500 lines (excluding machine-generated code).
2) Avoid methods with more than 200 lines.
3) Lines should not exceed 150 characters.
4) Avoid defining multiple types in the same file. You can define nested classes in the same file.

2 Coding Practices

2.1 Qualitative Directives

1) Always write readable code.
2) Detect mistakes as soon as possible.
3) Do not pre-optimize, that is optimizing with lack of concrete measurements or explicit need.
4) Do not optimize.
5) Avoid comments that explain the obvious. Code should be self-explanatory. Good code with readable variables and method names should not require comments.
6) Document only operational assumptions, algorithm insights and so on.
7) Avoid method-level documentation.
   a) Use extensive external documentation for API documentation.
   b) Use method-level comments only as tool tips for other developers.
8) All comments should pass spell checking.
9) Avoid passive voice in messages.
10) Assert every assumption. On average, every 6th line is an assertion:

```
using System.Diagnostics;

object GetObject()
{...}

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

11) Walk through every line of code in a “white box” testing manner.
12) Do not use late-binding invocation when early-binding is possible.
13) Use application logging and tracing liberally. You cannot log too much.
14) Do not suppress compiler warnings or errors that agree with this standard. Fix the code instead. Suppress those that disagree.

2.2 Variables, Constants, and Arguments

1) Always use C# predefined types rather than the aliases in the **`System`** namespace. For example:

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

2) Except for 0 and 1, never hard-code a numeric value; declare a constant instead.

3) Use the **`const`** directive only on natural constants such as number of days in a week.
4) Avoid using **`const`** on read-only variables. For that, use the **`readonly`** directive:

```
class MyClass
{
   public const int DaysInWeek = 7;
   public readonly int Number;

   public MyClass(int someValue)
   {
      Number = someValue;
   }
}
```

5) Only use enumerations to define concepts that seldom change, such as well-known status codes.
6) Prefer overloading to default parameters:

```
//Avoid:
void MyMethod(int number = 123)
{...}

//Correct:
void MyMethod()
{
   MyMethod(123);
}
void MyMethod(int number)
{...}
```

7) 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)
{...}
```

8) Use unsigned types for values that cannot be negative.
9) Use the smallest types possible, such as **`short`** for numbers -32,768 to 32,767.

2.3 Interfaces

1) Always use interfaces. See Chapters 1 and 3 in Programming .NET Components 2nd Edition.
2) Avoid interfaces with one member.
3) Strive to have three to five members per interface.

4) Do not have more than 20 members per interface. Twelve is probably the practical limit.
5) Avoid properties as interface members.
6) Avoid events (via delegates) as interface members.
7) Provide an interface for all abstract classes.
8) Expose interfaces on class hierarchies.
9) Always use explicit interface implementation.
10) Do not define constraints in generic interfaces. Replace interface-level constraints with strong-typing:

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

11) Do not define method-specific constraints in interfaces.
12) 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();
   }
}
```

13) Never assume a type supports an interface. Defensively query for that interface:

```
SomeType obj1 = ...;
IMyInterface obj2;

obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
   obj2.Method1();
}
else
{
   //Handle error in interface query
}
```

2.4 Delegates and Events

1) Use delegate inference instead of explicit delegate instantiation:

```
delegate void SomeDelegate();

void MyMethod()
{...}
//Avoid
SomeDelegate someDelegate = new SomeDelegate(MyMethod);

//Correct
SomeDelegate someDelegate = MyMethod;
```

2) Always check a delegate for **`null`** before invoking it. Consider initializing the delegate with an empty anonymous method:

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

3) Do not provide public event member variables. Use event accessors instead:

```
delegate void SomeDelegate(...);

//Do not:
class MyPublisher
{
   public SomeDelegate SomeEvent;
}
//Correct:
class MyPublisher
{
   SomeDelegate m_SomeEvent;

   public event SomeDelegate SomeEvent
   {
      add
      {
         m_SomeEvent += value;
      }
      remove
      {
         m_SomeEvent -= value;
      }
   }
}
```

4) Avoid defining event-handling delegates. Use **`EventHandler<T>`** or **`GenericEventHandler`** instead. See Chapter 6 of Programming .NET Components 2nd Edition.

5) Avoid raising events explicitly. Use **`EventsHelper`** to publish events defensively. See Chapters 6-8 of Programming .NET Components 2nd Edition.

6) Do not define generic constraints in delegates.

2.5 Exceptions

1) Catch only exceptions for which you have explicit handling.
2) Do not throw **`System.Exception`**.
3) If a **`catch`** statement throws an exception, always rethrow the original exception (or another exception constructed from the original exception) to maintain the stack location of the original error:

```
catch(Exception exception)
{
   Trace.WriteLine(exception.Message);
   throw;
}
```

4) Avoid tampering with exception handling using the Exception window (Debug|Exceptions).
5) Avoid error codes as method return values.
6) Avoid defining custom exception classes.
7) When defining custom exceptions:
   a) Derive custom exceptions from **`Exception`**.
   b) Provide custom serialization.

2.6 Construction and Initialization

1 Always provide the static constructor to initialize static member variables.

2 Avoid inline members initialization. Group all initialization in constructors:

```
//Avoid
class MyClass
{
   int m_Number = 123;
   static string m_Name = "Juval";
}
//Correct:
class MyClass
{
   int m_Number;
   static string m_Name;

   public MyClass()
   {
      m_Number = 123;
   }
   static MyClass()
   {
      m_Name = "Juval";
   }
}
```

3 With inline collection initialization, avoid more than a handful of items. The likely number is one item, and the maximum should be five:

```
//Avoid:
int[] numbers = {1,2,3,4,5,6,7,8,9,10,11,1,2,3,4,5,6,7};
//Better:
int[] numbers = {1,2,3};
//Best:
int[] numbers = {1};
```

4 Always explicitly initialize a collection of reference types. For example:

```
class MyClass
{}
int size = 100;
MyClass[] array = new MyClass[size];
for(int index = 0; index < size; index++)
{
   array[index] = new MyClass();
}
```

5 Avoid default **new** operator as shorthand for default constructor of the declared type:

```
class MyClass
{
   public MyClass()
   {...}
}

//Avoid:
MyClass myClass = new();

//Correct:
MyClass myClass = new MyClass();
```

6 Avoid all forms of object initializers:

```
//Avoid:
class MyClass
{
   public int Number;
}

MyClass myClass = new MyClass()
                  {
                     Number = 123
                  };
//Correct:
class MyClass
{
   int m_Number;

   public MyClass(int number)
   {
      m_Number = number;
   }
}
MyClass myClass = new MyClass(123);
```

7 Do not program against specific numeric values of enumerations.

8 Avoid providing explicit values for enums unless they are integer powers of 2, in which case add the **Flags** attribute, for bit masking:

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

9 Avoid specifying a type for an enum.

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

2.7 General

1) Avoid multiple **Main()** methods in a single assembly.
2) Minimize code in application assembly (EXE assembly or where **Main()** resides). Use class libraries instead to contain business logic.
3) Never use Top-Level Statements.
4) Never use friend assemblies in business code; doing so increases inter-assembly coupling. Use friend assemblies in framework code but only when absolutely necessary.
5) Avoid code that relies on an assembly running from a particular location.
6) Avoid methods with more than 5 arguments. Use structures or classes for passing multiple arguments.
7) Avoid manually editing any machine-generated code.
   a) When 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 using the **new** inheritance qualifier. Use **override** instead.
9) Never use unsafe code, except when using interop.
10) Avoid the ternary conditional operator.

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

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

12) Avoid changing mutable reference type arguments inside methods:

```
class MyClass
{
   public int Number;
}
//Avoid
void MyMethod(MyClass argument)
{
   argument.Number = 4;
}
```

13) Always use zero-based arrays.

14) Use zero-based indexes with indexed collections.

15) Prefer empty collections to null references:

```
//Avoid:
int[] MyMethod()
{
   return null;
}
//Correct:
int[] MyMethod()
{
   return new int[0];
}
```

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

17) If a class or a method offers both generic and non-generic flavors, always prefer using the generic flavor.

18) Avoid **`IEnumerable<T>`** in service interfaces and data contracts. Use arrays instead.

19) Prefer decimal in all floating-point types:

```
//Avoid:
float  number1 = 3.14f;
double number2 = 3.14;

//Correct:
decimal number3 = 3.14m;
```

20) 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)
{...}
```

21) 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;
   }
}
```

22) Never hardcode strings that end users may see. Use resources instead.

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

24) Use **`String.Empty`** instead of `""`:

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

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

25) When building a long string, use **`StringBuilder`**, not **`string`**.

26) Avoid providing methods on structures.

    a) Use parameterized constructors.

    b) Overload operators as required.

27) Do not use **`GC.AddMemoryPressure()`**.

28) Do not rely on **`HandleCollector`**.

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

30) Always use a full scope with a **`using`** statement, and never the **`using`** declaration:

```
//Never:
using(...);

//Always:
using(...)
{...}
```

31) Avoid function calls in Boolean conditional statements. Assign to local variables and check on them:

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

32) Avoid compound statements:

```
int Foo()
{...}

void Bar(int number)
{...}

class MyClass
{
   public MyClass(int number)
   {...}
}

//Avoid:
Bar(Foo());

//Correct:
int number = Foo();
Bar(number);

//Avoid:
return Foo();

//Correct:
int number = Foo();
return number;

//Avoid:
MyClass myClass = new MyClass(Foo());

//Correct:
int number = Foo();
MyClass myClass = new MyClass(number);
```

33) 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;}
}
```

34) Never use **`goto`** unless in a **`switch`** statement fall-through.

35) Always have a **`default`** case in a **`switch`** statement that asserts or throws:

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

      Trace.WriteLine("Case 2:");
      break;
   }
   default:
   {
      Debug.Assert(false);
      break;
   }
}
```

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

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

37) Do not use the **`this`** reference unless invoking another constructor from within a constructor, or querying yourself for an interface reference:

```
//Example of proper use of 'this'
interface IMyInterface
{...}
class MyClass : IMyInterface
{
   public MyClass(string message)
   {}
   public MyClass() : this("Hello")
   {}
   void SomeMethod()
   {
      IMyInterface myInterface = this;
   }
   ...
}
```

38) Do not use the **`base`** word to access base class members unless resolving a conflict with a subclass member of the same name, or when invoking a base class constructor:

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

39) Only use **`var`** with anonymous types. Always explicitly declare the type for readability and correctness:

```
//Avoid:
var result = GetResult();

//Correct:
int result = GetResult();

//Avoid:
var number = 3.14;

//Correct:
float number = 3.14;
```

40) Do not use anonymous types.

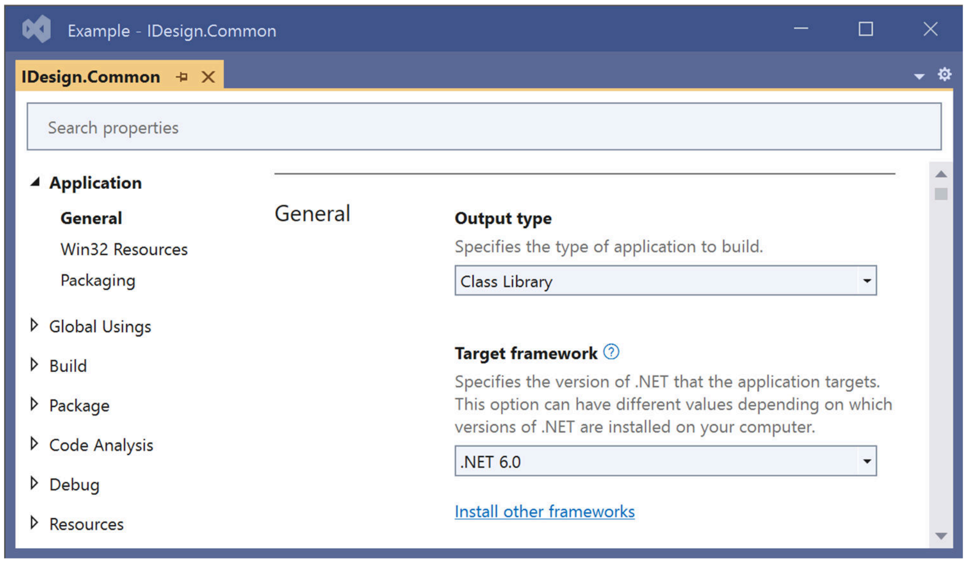
41) Do not use complex LINQ statements.

42) 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. Include parentheses:

```
delegate void SomeDelegate(string someString);

void MyMethod(SomeDelegate someDelegate)
{...}

void DoSomething()
{...}

//Avoid
MyMethod(name =>{DoSomething();Trace.WriteLine(name);});

//Correct:
MyMethod((string name)=> Trace.WriteLine(name));
```

3 Project Settings and Project Structure

1. Prefer to include solution related files (e.g., *editorconfig*) in the root of the repository in the solution's Solution Items folder.
2. Always use a *Directory.Build.props* file in the root of your repository to enforce policy across all projects.
3. Do not use the Visual Studio Project Properties window to change values when using a *Directory.Build.props* file. Set the project properties through editing either the *Directory.Build.props* file or a specific project file.
4. Always target the current long-term support (LTS) release required for your solution, unlike the default which will give you the latest:



Output type

Specifies the type of application to build.

Class Library

Target framework

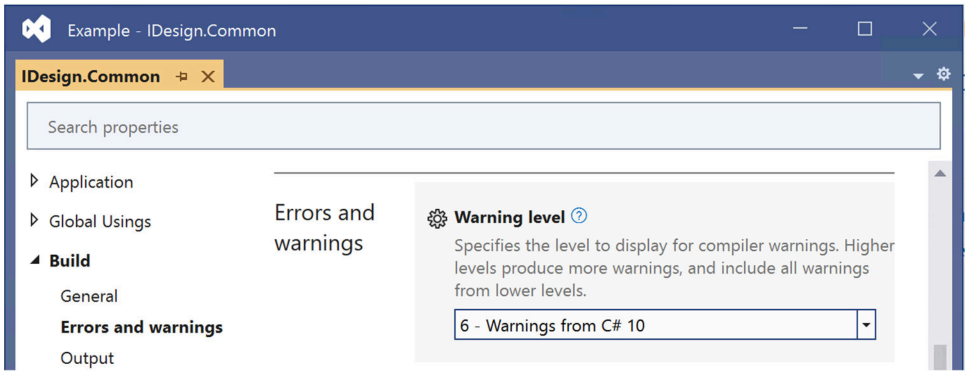
Specifies the version of .NET that the application targets. This option can have different values depending on which versions of .NET are installed on your computer.

.NET 6.0

Install other frameworks

```
<Project>
   <PropertyGroup>
      <TargetFramework>net6.0</TargetFramework>
   </PropertyGroup>
</Project>
```

5. Always build your project with the highest warning appropriate for your C# language version:



Warning level

Specifies the level to display for compiler warnings. Higher levels produce more warnings, and include all warnings from lower levels.

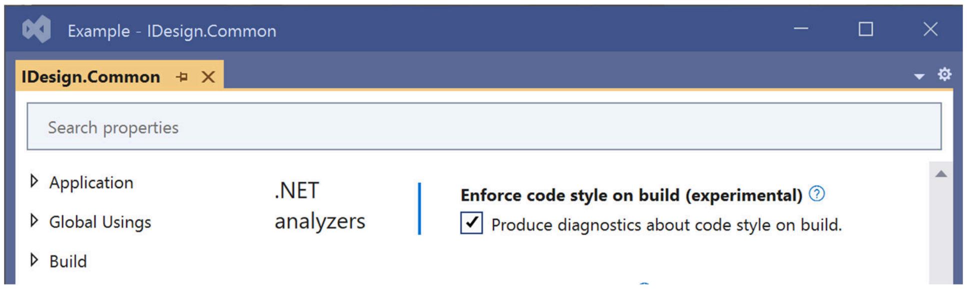
6 - Warnings from C# 10

```
<Project>
   <PropertyGroup>
      <WarningLevel>6</WarningLevel>
   </PropertyGroup>
</Project>
```

6. Always explicitly specify language version. Match the Target Framework's default language version:

```
<Project>
   <PropertyGroup>
      <LangVersion>10.0</LangVersion>
   </PropertyGroup>
</Project>
```

7. Always build your project using **`EnforceCodeStyleInBuild`** set to **`true`**:

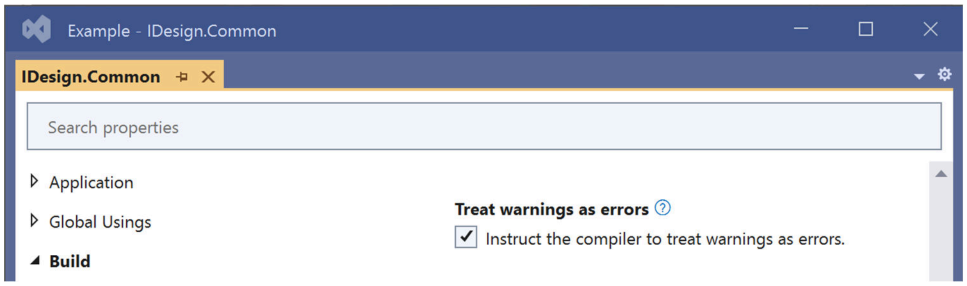


Enforce code style on build (experimental)

☑ Produce diagnostics about code style on build.

```
<Project>
   <PropertyGroup>
      <EnforceCodeStyleInBuild>True</EnforceCodeStyleInBuild>
   </PropertyGroup>
</Project>
```

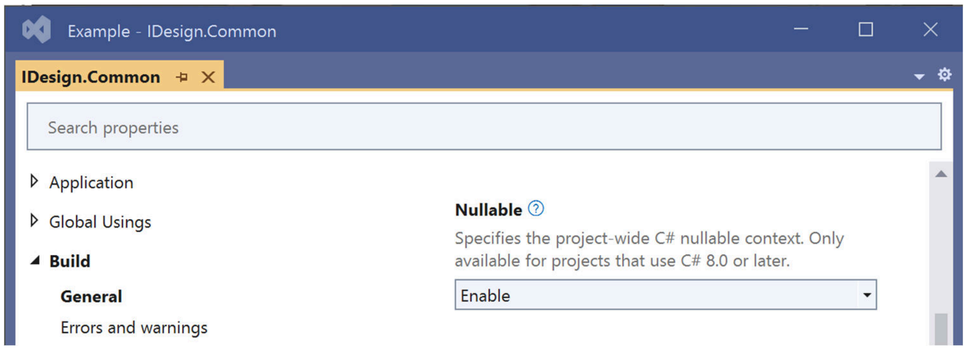
8. Always include *.editorconfig* to ensure uniform code style across projects.
9. Treat warnings as errors in the Release build (note that this is not the default of Visual Studio). Although it is optional, this standard recommends treating warnings as errors in Debug builds as well:



Treat warnings as errors
☑ Instruct the compiler to treat warnings as errors.

```
<Project>
   <PropertyGroup>
      <TreatWarningsAsErrors>True</TreatWarningsAsErrors>
   </PropertyGroup>
</Project>
```

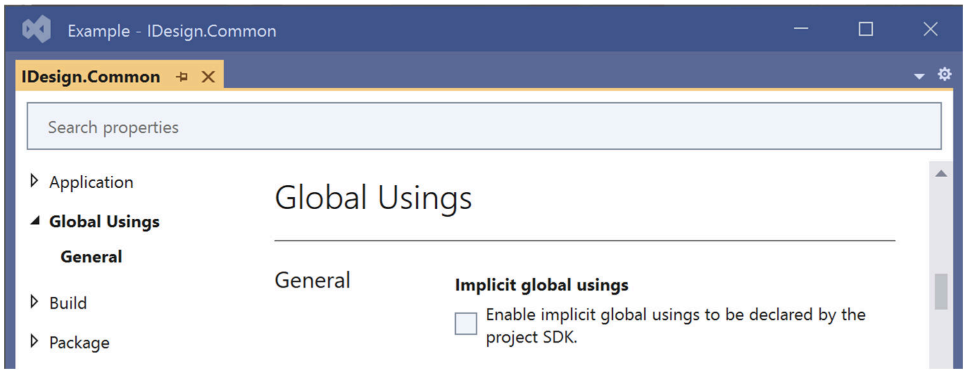
10. Always prefer Nullable Contexts:



Nullable
Specifies the project-wide C# nullable context. Only available for projects that use C# 8.0 or later.
Enable

```
<Project>
   <PropertyGroup>
      <Nullable>enable</Nullable>
   </PropertyGroup>
</Project>
```

11. Never use implicit global usings:



Global Usings

General

Implicit global usings

☐ Enable implicit global usings to be declared by the project SDK.

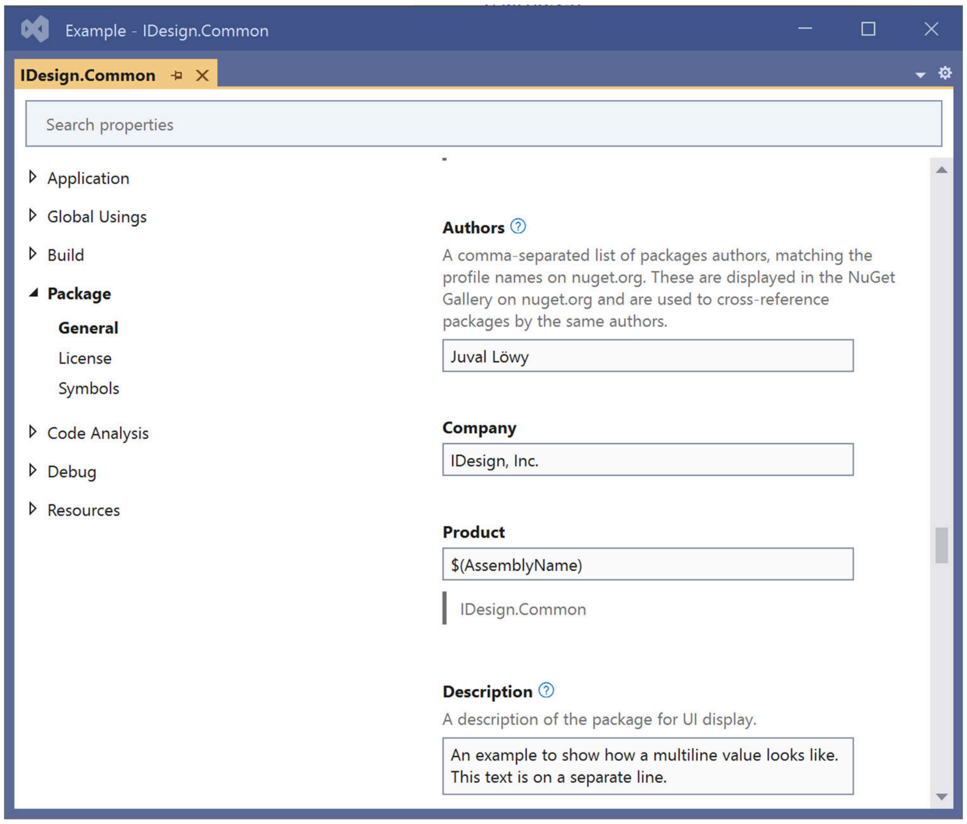
```
<Project>
   <PropertyGroup>
      <ImplicitUsings>disable</ImplicitUsings>
   </PropertyGroup>
</Project>
```

12. Always explicitly state your .NET SDK version by including a *global.json* file in the root directory. Always roll forward to the latest feature:

```
{
   "sdk":
   {
      "version": "6.0.300",
      "rollForward": "latestFeature"
   }
}
```

13. Avoid explicit preprocessor definitions (**`#define`**). Use the project settings for defining conditional compilation constants.
14. Put Assembly Attributes in the project file or *Directory.Build.props* file.

15. Populate all Package properties such as authors, company name, description, and copyright notice using the *Directory.Build.props* file:



Authors

A comma-separated list of packages authors, matching the profile names on nuget.org. These are displayed in the NuGet Gallery on nuget.org and are used to cross-reference packages by the same authors.

Juval Löwy

Company

IDesign, Inc.

Product

$(AssemblyName)

IDesign.Common

Description

A description of the package for UI display.

An example to show how a multiline value looks like.
This text is on a separate line.

```
<Project>
   <PropertyGroup>
      <Authors>Juval Löwy</Authors>
      <Company>IDesign, Inc.</Company>
      <Description>An example to show what a multiline value looks like.
This text is on a separate line.</Description>
      <Copyright>© 2023 IDesign Inc, All rights reserved.</Copyright>
   </PropertyGroup>
</Project>
```

16. Use relative paths for all assembly references in the same solution.
17. Avoid single-file publishing.
18. Always use a SemVer 2.0 compatible version number.
19. Always use build tools to set the version assembly attribute at build time. This value is the default value for package version, assembly version and file version:

```
dotnet build -p:Version=1.2.3.4
```

20. Never set package version, assembly version and file version manually. Use Central Package Management and its *Directory.Packages.props* file to prevent unconsolidated Package References.
21. Consider a private NuGet repository to curate available packages.
22. Always include *nuget.config* to explicitly specify package sources and use package source mapping:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <packageSources>
      <add key = "IDesign.net" value = "https://idesign.net/packages/" />
   </packageSources>
   <packageSourceMapping>
      <packageSource key = "IDesign.net">
         <package pattern =" IDesign.*" />
      </packageSource>
   </packageSourceMapping>
</configuration>
```

23. Modify the IDE default project structure to comply with your project standard layout and apply uniform structure for project folders and files.
24. Always include debug symbols in a Release build. Do not change the default.
25. Always sign your assemblies, including the client applications.
26. Use password-protected keys when generating key-pairs for strong naming assemblies.
27. Consider using a cryptographic container for signing keys rather than distributing files.
28. Consider delay-signing assemblies during development.
29. Consider digitally signing (SignTool) all assemblies.
30. Use a certificate from a trusted Certificate Authority for digitally signing assemblies.
31. Apply encryption and security protection on application configuration files.

4 Asynchronous & Parallel Programming

1. Prefer the task-based asynchronous pattern (TAP) to manual multithreading.
2. Never use TAP for CPU-bound workloads.
    a) Only use TAP in multichannel I/O-bound scenarios or in scenarios that have single-threaded affinity, such as a UI application.
3. When using the **`Async`** suffix to TAP methods, always apply regardless of whether your TAP method uses **`async`**/**`await`**.
4. Never wrap synchronous code in asynchronous wrapper methods.
5. Only apply the **`async`** keyword to methods that **`await`** asynchronous calls.
6. Never return **`void`** from an **`async`** method. Always use Task instead.
7. Prefer **`async`**/**`await`** language support instead of using **`Wait()`** or **`Result`** to wait on **`Task`** completion. This promotes efficient thread usage in I/O-bound scenarios.
    a) When using **`async`**/**`await`**, always apply the pattern consistently throughout your programming flows.

```
//Avoid:
Task MyMethod1Async()
{...}
Task<string> MyMethod2Async()
{...}
void MyCallingMethod()
{
   MyMethod1Async().Wait();
   string result = MyMethod2Async().Result;
}

//Correct:
async Task MyMethod1Async()
{...}
async Task<string> MyMethod2Async()
{
   await MyMethod1Async()
   ...
}
async Task MyCallingMethodAsync()
{
   string result = await MyMethod2Async();
}
```

8. Never use blocking calls within **async/await** programming flows or continuation deadlock may result:

```
async Task MyMethod1Async()
{...}
async Task<string> MyMethod2Async()
{...}
async Task MyCallingMethodAsync()
{
   //Never:
   MyMethod1Async().Wait();
   string result = MyMethod2Async().Result;

   //Correct:
   await MyMethod1Async();
   result = await MyMethod2Async();
}
```

9. Never assume a continuation will resume on the same thread as the antecedent Task.

   a) In scenarios that do not have single-threaded affinity, this includes the code after an **await** statement.

10. Unless your asynchronous design requires it, prefer to use **async/await** language support instead of explicit continuations:

```
async Task MyMethodAsync()
{...}
async Task MyCallingMethod()
{
   //Avoid:
   Task task = MyMethodAsync();
   await task.ContinueWith((Task result)=>//Continuation code here);

   //Correct:
   await MyMethodAsync();
   //Continuation code here
   ...
}
```

11. In scenarios that have single-threaded affinity, always use **async/await** instead of explicit continuations:

```
async Task MyMethodAsync()
{...}
async Task MyUIMethod()
{
   //On UI thread here
   //Avoid:
   Task task = MyMethodAsync();
   await task.ContinueWith((Task result)=>//Not on UI here);
   //Correct:
   await MyMethodAsync(); //Sync context picked up here
   //On UI thread here
   ...
}
```

12. When using explicit continuations, never use fluent programming:

```
async Task MyMethodAsync()
{...}
async Task MyCallingMethod()
{
   //Never:
   await MyMethodAsync().ContinueWith((Task result)=>...);

   //Correct:
   Task task = MyMethodAsync();
   await task.ContinueWith((Task result)=>...);
}
```

13. When using explicit continuations, always consolidate your result processing:

```
void MyResultProcessing(Task result)
{
   //Result processing here
}
async Task MyMethodAsync()
{}
async Task MyCallingMethodAsync()
{
   Task task = MyMethodAsync();

   //Avoid:
   await task.ContinueWith((Task result)=>//Result processing here);

   //Correct:
   await task.ContinueWith((Task result)=>MyResultProcessing(result));
}
```

14. When using explicit continuations, always consolidate exception management policy in one place.
    a) Always check **`Task.IsFaulted`** first, because **`Task.IsCompleted`** will also be **`true`**.
    b) Alternately, check **`Task.Status.Faulted`**.
    c) Optionally, restrict continuations specifically for exception management policy using **`TaskContinuationOptions.OnlyOnFaulted`**.

```
void MyExceptionPolicy(AggregateException exceptions)
{
   //Exception management here
}
void MyResultProcessing(Task result)
{
   if(result.IsFaulted)
   {
      MyExceptionPolicy(result.Exception);
   }
   ...
```

```
}
async Task MyMethodAsync()
{
   //Avoid:
   try
   {...}
   catch(Exception exception)
   {//Exception management here}
}
async Task MyCallingMethodAsync()
{
   Task task = Task.Run(()=>
                        {
                           //Avoid:
                           try
                           {...}
                           catch(Exception exception)
                           {//Exception management here}
                         };

   //Correct:
   Task task = MyMethodAsync();
   await task.ContinueWith((Task result)=>MyResultProcessing(result));

   //Optionally:
   Action<Task> faultHandler = (Task result)=>
                               {
                                  MyExceptionPolicy(result.Exception);
                               };
   await task.ContinueWith(faultHandler,TaskContinuationOptions.OnlyOnFaulted);
}
```

15. When using explicit continuations where a **SynchronizationContext** exists, always change the default **TaskScheduler** of the continuation.

    a) Use **TaskScheduler.FromCurrentSynchronizationContext()** to obtain a **TaskScheduler** based on the parent thread's current **SynchronizationContext**.

```
async Task MyMethodAsync()
{...}
async Task MyCallingMethod()
{
   //Parent sync context here
   Debug.Assert(SynchronizationContext.Current != null);

   Task task = MyMethodAsync();

   //Never
   await task.ContinueWith((Task result)=>//No sync context here);
   //Correct:
   Action<Task> continuation = (Task result)=>
                               {
                                  //Parent sync context here
                               };

   await task.ContinueWith(result,
                               TaskScheduler.FromCurrentSynchronizationContext());
}
```

16. Never use **async/await** programming flows as a replacement for Task Parallel Library (TPL) parallel programming techniques.
    a) Instead, always use the parallelism concepts within the TPL, such as **Parallel.ForEach()**, **Parallel.Invoke()** or **Task.When<…>()** variants.
17. Never use **Task.WaitAll()** variants.
    a) Avoid using **Task.WhenAny()** variants.
    b) Prefer to use **Task.WhenAll()** variants.
18. Never expect partial results from **Task.WhenAll()**.
19. When using **Parallel.ForEach()** or **Parallel.Invoke()**, always catch **AggregateException**. It contains exceptions from all iterations with **TaskStatus.Faulted**.

```
void MyExceptionPolicy(AggregateException exceptions)
{...}
void MyCallingMethod()
{
   string[] source = ...;
   try
   {
      Action<string> action = (string element)=>
                              {
                                 ...
                              };
      Parallel.ForEach(source,action);
   }
   catch(AggregateException exception)
   {
      MyExceptionPolicy(exception);
   }
}
```

20. Always interrogate the inner exceptions (there may be multiple) from **AggregateException**.

```
void MyExceptionPolicy(AggregateException exception)
{
   foreach(Exception innerException in exception.InnerExceptions)
   {...}
}
```

21. Never use **async/await** programming flows within **Parallel.ForEach()** or **Parallel.Invoke()**.
    a) The parallelism concept (TPL) returns immediately leaving continuations to execute unobserved.
    b) Always explicitly wait upon the returned **Task**.

```
Task MyMethodAsync(string element)
{...}
void MyCallingMethod()
{
   string[] source = ...;

   //Never:
   Parallel.ForEach(source,async (string element)=>await MyMethodAsync(element));

   //Correct:
   Parallel.ForEach(source,(string element)=>MyMethodAsync(element).Wait());
}
```

22. Never parallelize the modification of collection concepts, concurrent or otherwise.
    a) Always modify collection concepts using synchronous techniques.

```
void MyCallingMethod()
{
   string[] source = ...;
   List<string> values = new List<string>();

   //Never:
   Parallel.ForEach(source,(string element)=>values.Add(...));

   //Correct:
   foreach(string element in source)
   {
      values.Add(...);
   }
}
```

23. When accumulating results from parallelized execution, never use **Parallel.ForEach()**. Instead, always use **Task.WhenAll()**.
    a) Only parallelize execution that resolves to multichannel I/O-bound workloads, such as parallelizing database or service calls.

```
async Task<string> MyServiceCallAsync(string data)
{...}
async Task MyCallingMethodAsync()
{
   string[] source = ...;

   //Never:
   ConcurrentDictionary<...> results = new ConcurrentDictionary<...>();
   Parallel.ForEach(source,(string element)=>
                            {
                               string result = MyServiceCallAsync(element).Result;
                               results.AddOrUpdate(element,result,...));
```

```
                           });

   //Correct:
   List<Task<string>> tasks = new List<Task<string>>();
   foreach(string element in source)
   {
      Task<string> task = MyServiceCallAsync(element);
      tasks.Add(task);
   }
   Task<string[]> whenAllTask = Task.WhenAll(tasks);
   string[] results = await whenAllTask;
}
```

24. Prefer **`Task.Run()`** to **`Task.Start()`** or **`TaskFactory.StartNew()`**.
25. When using **`Task.Run()`**, always **`await`** or **`return`** the inner **`Task`**. Otherwise, the inner **`Task`** completes immediately leaving your tasks and any continuations to execute unobserved.

```
Task MyLongRunningMethodAsync()
{...}
void MyCallingMethod()
{
   //Never:
   Task task1 = Task.Run(()=>
                      {
                         MyLongRunningMethodAsync();
                      });

   //Correct:
   Task task1 = Task.Run(()=>
                      {
                         Task result = MyLongRunningMethodAsync();
                         return result;
                      });
   Task task2 = Task.Run(async ()=>
                            {
                               await MyLongRunningMethodAsync();
                            });
}
```

26. In scenarios that have thread affinity, always defer long running workloads for asynchronous execution using **`Task.Run()`**.
    a) Never use **`Task.Run()`** within a long running method.
    b) Never call **`Wait()`** or **`Result`** on or **`await`** the task returned from **`Task.Run()`** or any continuations you provide. Doing so blocks the parent thread.
    c) If your design requires it, marshal your continuation back to the parent thread using **`TaskScheduler.FromCurrentSynchronizationContext()`**

27. Only use **`Task.Run()`** within **`async`/`await`** programming flows to parallelize multiple tasks that may perform lengthy preparation before asynchronous invocation.
    a) Always return the un-awaited **`Task`** from **`Task.Run()`**.
    b) Use only in conjunction with the **`Task.WhenAll()`** parallelization techniques shown previously.
    c) In all other scenarios, collect the tasks directly without using **`Task.Run()`**.

```
void MyLengthyPrep()
{...}
async Task<string> MyServiceCallAsync(string data)
{
   MyLengthyPrep();
   string result = await //async call here;
   return result;
}
async Task MyCallingMethodAsync()
{
   string[] source = ...;

   foreach(string element in source)
   {
      //Never:
      string result = await MyServiceCallAsync(element);
      result = await Task.Run(()=>return MyServiceCallAsync(element));
      result = await Task.Run(async ()=>await MyServiceCallAsync(element));
   }

   //Correct:
   List<Task<string>> tasks = new List<Task<string>>();
   foreach(string element in source)
   {
      Task<string> task = Task.Run(async ()=>await MyServiceCallAsync(element));
      tasks.Add(task);
   }
   Task<string[]> whenAllTask = Task.WhenAll(tasks);
   string[] results = await whenAllTask;
}
```

28. Prefer not to await **`Task.WhenAll()`** directly. Instead create a result **`Task`** and await it.
    a) Do not attempt to catch the exception of the awaited result from **`Task.WhenAll()`** as it will only contain the first exception thrown.
    b) Instead, within the **`catch`** block interrogate the exception property of the result **`Task`**. It will contain all exceptions.
    c) Always first check if there were inner exceptions.

```
async Task MyCallingMethodAsync()
{
   //Avoid:
   try
   {
      await Task.WhenAll(...);
   }
   catch(Exception exception)
   {...}

   //Avoid:
   try
   {
      ...
      Task whenAllAsync = Task.WhenAll(...);
      await whenAllAsync;
   }
   catch(Exception exception)
   {...}

   //Correct:
   try
   {
      ...
      Task whenAllAsync = Task.WhenAll(...);
      await whenAllAsync;
   }
   catch
   {
      AggregateException exception = whenAllAsync.Exception;
      MyExceptionPolicy(exception);
   }
}
```

29. When using TAP with TPL parallelism concepts, always support **`Task`** cancellation by creating asynchronous methods that take a **`CancellationToken`** as a parameter.
    a) Never expect **`Task`** cancellation using a **`CancellationToken`** to abort a running or blocked thread or a **`Task`** awaiting continuation.
    b) Always explicitly support **`Task`** cancellation as your asynchronous design requires.

```
Task MyMethodAsync(...,CancellationToken token)
{
   if(token.IsCancellationRequested == false)
   {...}
}
Task MyLongRunningMethodAsync(...,CancellationToken token)
{
   while(token.IsCancellationRequested == false)
   {...}
}
```

30. Always honor support for **`Task`** cancellation in your code for 3rd party libraries you use by passing through any cancellation tokens they provide:

```
abstract class ThirdPartyBase
{
   protected virtual async Task ThirdPartyMethodAsync(...,CancellationToken token)
   {...}
}
class MySubClass : ThirdPartyBase
{
   async Task MyMethodAsync(...,CancellationToken token)
   {...}
   protected override async Task ThirdPartyMethodAsync(...,CancellationToken token)
   {
      await MyMethodAsync(...,token);
      ...
   }
}
```

31. Always properly control **`Task`** cancellation by creating a **`CancellationTokenSource`** and managing the cancellation token if one does not exist:

```
Task MyLongRunningMethodAsync(..., CancellationToken token)
{...}
void MyCallingMethod()
{
   CancellationTokenSource tokenSource = new CancellationTokenSource();
   CancellationToken token = tokenSource.Token;

   Task.Run(()=>return MyLongRunningMethodAsync(...,token));
   ...
   tokenSource.Cancel();
}
```

32. When using **`Parallel.ForEach()`**, prefer to support **`Task`** cancellation by using **`ParallelOptions`** to set a **`CancellationToken`** for all iterations.

    a) Always explicitly support **`Task`** cancellation by assessing **`ParallelLoopState.ShouldExitCurrentIteration`** as your asynchronous design requires.

```
void MyCallingMethod()
{
   string[] source = ...;

   CancellationTokenSource tokenSource = new CancellationTokenSource();
   CancellationToken token = tokenSource.Token;

   ParallelOptions options = new ParallelOptions();
   options.CancellationToken = token;

   Action<string,ParallelLoopState> action = (string element,
                                              ParallelLoopState state)=>
                                              {
                                                 while(
                                              state.ShouldExitCurrentIteration == false)
                                                   {
                                                      ...
                                                   }
                                                };
   Parallel.ForEach(source,options,action);
   ...
   tokenSource.Cancel();
}
```

33. When using **Task** cancellation with **Parallel.ForEach()**, always add an explicit **catch** block for **OperationCanceledException**.
    a) **Parallel.ForEach()** throws **OperationCanceledException** when you cancel all iterations.

```
void MyCallingMethod()
{
   string[] source = ...;
   try
   {
      Action<string,ParallelLoopState> action = (string element,
                                                  ParallelLoopState state)=>
                                                {
                                                   while(
                                              state.ShouldExitCurrentIteration == false)
                                                   {
                                                      ...
                                                   }
                                                };
      Parallel.ForEach(source,options,action);
   }
   catch(OperationCanceledException exception)
   {...}
   catch(AggregateException exception)
   {...}
}
```

34. When using **Task** cancellation with **Task.WhenAll()**, always add an explicit **catch** block for **TaskCanceledException**.
    a) **Task.WhenAll()** throws **TaskCanceledException** when you cancel all tasks.

```
async Task MyCallingMethodAsync()
{
   try
   {
      ...
      Task whenAllAsync = Task.WhenAll(...);
      await whenAllAsync;
   }
   catch(TaskCanceledException exception)
   {...}
   catch
   {...}
}
```

5 Multithreading

1. Manage asynchronous call completion on a callback method. Do not wait, poll, or block for completion.
2. Always name your threads. The debugger Threads window name traces the name, making debug sessions more productive:

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

3. Do not call **`Suspend()`** or **`Resume()`** on a thread.
4. Do not call **`Thread.Sleep()`**, except in the following conditions:
   a) **`Thread.Sleep(0)`** is an acceptable technique to force a context switch.
   b) **`Thread.Sleep()`** is acceptable in testing or simulation code.
5. Do not call **`Thread.SpinWait()`**.
6. 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.
7. Avoid explicitly setting thread priority to control execution. You can lower thread priority based on task semantics, such as below normal (**`ThreadPriority.BelowNormal`**) for a screen saver.
8. Do not read the value of the **`ThreadState`** property. Use **`Thread.IsAlive`** to determine whether the thread is dead or alive.
9. 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.
10. Do not use thread local storage unless you can guarantee thread affinity.
11. Do not call **`Thread.MemoryBarrier()`**.
12. Never call **`Thread.Join()`** without checking not joining own thread:

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

13. Always use the **`lock()`** statement rather than explicit **`Monitor`** manipulation.

14. Always encapsulate the **`lock()`** statement inside the object it protects:

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

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

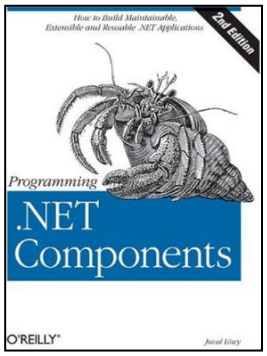
6 Transactions

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 0 (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 constructor of **`TransactionScope`**.
8. Always catch and discard exceptions thrown by a transaction scope that is set to **`TransactionScopeOption.Suppress`**.

Resources

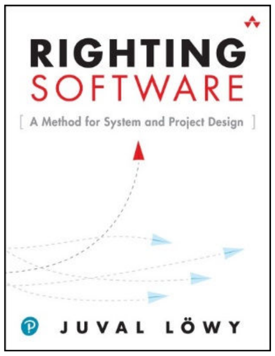
Programming .NET Components 2nd Edition

By Juval Löwy, O'Reilly 2005


How to Build Maintainable, Extensible and Reusable .NET Applications
2nd Edition
Programming
.NET Components
O'REILLY
Juval Löwy

Righting Software 1st Edition

By Juval Löwy, Addison-Wesley, 2020



The Architect's Master Class

The Architect's Master Class is the ultimate resource for the professional architect. The class shows how to take an active leadership role and is often referred to as a career-changing event. Alumni of the class are the architects of some of the most well-known companies and projects around the world. While the class shows how to design modern systems, it sets the focus on the 'why' and the rationale behind design decisions, often shedding light on poorly understood aspects. You will see relevant design guidelines, best practices, pitfalls, and the crucial process required of today's modern architects. Do not miss this unique opportunity to learn and improve your design skills with IDesign and share our passion for architecture and software engineering. More at www.idesign.net