C# Coding Standards for .NET

Changes for 1/05/2018

Author: Kevin Roach

Copyright Kevin Roach 2018

All Rights Reserved

Contents

[Revision 4](#_Toc502935405)

[1 Introduction 5](#_Toc502935406)

[1.1 Document Conventions 5](#_Toc502935407)

[1.1.1 Coloring & Emphasis 5](#_Toc502935408)

[2.1.2 Keywords 5](#_Toc502935409)

[2.2 Terminology & Definitions 5](#_Toc502935410)

[3 Framework Design Guidelines 6](#_Toc502935411)

[3.1 Capitalization Conventions 6](#_Toc502935412)

[3.1.1 Capitalization Rules 6](#_Toc502935413)

[3.1.2 Capitalizing Compound Words and Common Terms 7](#_Toc502935414)

[3.2 General Naming Conventions 7](#_Toc502935415)

[3.2.1 Word Choice 7](#_Toc502935416)

[3.2.2 Using Abbreviations and Acronyms 7](#_Toc502935417)

[3.2.3 Names of Namespaces 7](#_Toc502935418)

[3.3 Names of Classes, Structs, and Interfaces 8](#_Toc502935419)

[3.4 Names of Generic Type Parameters 8](#_Toc502935420)

[3.5 Names of Type Members 8](#_Toc502935421)

[3.5.1 Names of Methods 8](#_Toc502935422)

[3.5.2 Names of Properties 9](#_Toc502935423)

[3.5.3 Names of Fields 9](#_Toc502935424)

[3.6 Naming Parameters 9](#_Toc502935425)

[3.7 Coding Style 9](#_Toc502935426)

[3.8 Language Usage 10](#_Toc502935427)

[4 Writing clean code 10](#_Toc502935428)

[4.1 Poor Names 10](#_Toc502935429)

[4.1.1 Mysterious Names 10](#_Toc502935430)

[4.1.2 Meaningless Names 11](#_Toc502935431)

[4.1.3 Ambiguous Names 11](#_Toc502935432)

[4.1.4 Noisy Names 11](#_Toc502935433)

[4.2 Poor Method Signatures 11](#_Toc502935434)

[4.3 Long Parameter List 12](#_Toc502935435)

[4.4 Output Parameters 12](#_Toc502935436)

[4.5 Inline Variables 12](#_Toc502935437)

[4.6 Magic Numbers 13](#_Toc502935438)

[4.7 Nested Conditionals 13](#_Toc502935439)

[4.7.1 Ternary Operator 13](#_Toc502935440)

[4.7.2 Simplify true/false 14](#_Toc502935441)

[4.7.3 Early Exit 14](#_Toc502935442)

[4.7.4 Swap Order 14](#_Toc502935443)

[4.7.5 Switch Statements 15](#_Toc502935444)

[4.7.6 Long Methods 15](#_Toc502935445)

[5 Quick Summary 15](#_Toc502935446)

[5.1 Naming Guidelines 15](#_Toc502935447)

[5.2 Coding Style 16](#_Toc502935448)

[5.3 Variables & Types 16](#_Toc502935449)

[4.6 Object Composition 16](#_Toc502935450)

[6 C# remove code smells 17](#_Toc502935451)

[6.1 Null-Conditional Operator 17](#_Toc502935452)

[6.1.1 ? Operator 17](#_Toc502935453)

[6.1.2 ?? Operator 17](#_Toc502935454)

[6.1.3 String Interpolation 17](#_Toc502935455)

[6.2 Exception Filters 18](#_Toc502935456)

[6.2.1 When 18](#_Toc502935457)

[6.2.2 Throw Expressions 18](#_Toc502935458)

[7 References 19](#_Toc502935459)

# Revision

|  |  |  |
| --- | --- | --- |
| **Date** | **Change** | **By whom** |
| 05/30/2017 | Document Creation | Kevin Roach |
| 01/05/2018 | Updated documentation from C# 6 (.NET Framework 4.6) to C# 7.2 (.NET Core) | Kevin Roach |

# Introduction

This document describes rules and recommendations for developing applications and class libraries using the C# Language. The goal is to define guidelines to enforce consistent style and formatting between different developers for a consistent look and feel.

Specifically, this document covers Naming Conventions, Coding Style, Language Usage, Writing Clean Code, and best coding standards.

## Document Conventions

Much like the ensuing coding standards, this document requires standards in order to ensure clarity when stating the rules and guidelines. Certain conventions are used throughout this document to add emphasis.

Below are some of the common conventions used throughout this document.

* + 1. Coloring & Emphasis:
* Blue Text colored blue indicates a C# keyword or .NET type.
* **Bold** Text with additional emphasis to make it stand-out.
  + 1. Keywords:
* **✓ DO**  Emphasizes this rule must be enforced.
* **X DO NOT** Emphasizes this action must not happen.
* **X Avoid** Emphasizes that the action should be prevented, but some exceptions may exist.
* **Try** Emphasizes that the rule should be attempted whenever possible and appropriate.
* **Consider** Emphasizes that the rule should be attempted whenever possible and appropriate.
* **Example** Precedes text used to illustrate a rule or recommendation.

## Terminology & Definitions

The following terminology is referenced throughout this document:

**Self-Documenting Code**

Code that allegedly explains itself without the need of extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc.

**Access Modifier**

C# keywords public, protected, internal, and private declare the allowed code-accessibility of types and their members. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public.

**Camel Case**

A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized.

**Pascal Case**

A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized.

**Identifier**

A developer defined token used to uniquely name a declared object or object instance.

**Example**:public classMyClassNameIdentifier{ … }

**Magic Number**

A number that appears out of the blue and its unclear form the code what the number represents

# Framework Design Guidelines

## Capitalization Conventions

This section contains tables describing a high-level summary of the major standards covered in this document. These tables are not comprehensive, but give a quick glance at commonly referenced elements.

**✓ DO** use PascalCasing for all public member, type, and namespace names consisting of multiple words.

**✓ DO** use camelCasing for parameter names.

### Capitalization Rules

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Casing** | **Example** |
| Namespace | Pascal | namespace System.Security { ... } |
| Type | Pascal | public class StreamReader { ... } |
| Interface | Pascal | public interface IEnumerable { ... } |
| Method | Pascal | public class Object  {  public virtual string ToString();  } |
| Property | Pascal | public class String  {  public string AccountType { get; set; }  } |
| Field (class level) | Pascal | public class FormController  {  private int Age;  } |
| Parameter (local fields) | Camel | public class Convert  {  public static int ToInt32(string value);  } |
| Enum value | Pascal | public enum FileMode  {  Append  } |

An additional example, where the purple words are PascalCase and the orange are camelCase

public class Customer

{

private int Id;

public string Name { get; set; }

public void Charge(int amount)

{

var tax = 0;

}

}

### Capitalizing Compound Words and Common Terms

|  |  |  |
| --- | --- | --- |
| **Pascal** | **Camel** | **Not** |
| BitFlag | bitFlag | Bitflag |
| Callback | callback | CallBack |
| Canceled | canceled | Cancelled |
| DoNot | doNot | Don't |
| Email | email | EMail |
| Endpoint | endpoint | EndPoint |
| FileName | fileName | Filename |
| Gridline | gridline | GridLine |
| Hashtable | hashtable | HashTable |
| Id | id | ID |
| Indexes | indexes | Indices |
| LogOff | logOff | LogOut |
| LogOn | logOn | LogIn |
| Metadata | metadata | MetaData, metaData |
| Multipanel | multipanel | MultiPanel |
| Multiview | multiview | MultiView |
| Namespace | namespace | NameSpace |
| Ok | ok | OK |
| Pi | pi | PI |
| Placeholder | placeholder | PlaceHolder |
| SignIn | signIn | SignOn |
| SignOut | signOut | SignOff |
| UserName | userName | Username |
| WhiteSpace | whiteSpace | Whitespace |
| Writable | writable | Writeable |

## General Naming Conventions

### Word Choice

**X DO NOT** use underscores, hyphens, or any other non-alphanumeric characters.

**X DO NOT** use Hungarian notation.

**X AVOID** using identifiers that conflict with keywords of widely used programming languages.

### Using Abbreviations and Acronyms

**✓ DO** use semantically interesting names rather than language-specific keywords for type names.

*For example*, GetLength is a better name than GetInt.

**X DO NOT** use abbreviations or contractions as part of identifier names.

*For example*, use GetWindow rather than GetWin.

**X DO NOT** use any acronyms that are not widely accepted, and even if they are, only when necessary.

### Names of Namespaces

**✓ DO** prefix namespace names with a company name to prevent namespaces from different companies from having the same name.

**✓ CONSIDER** using plural namespace names where appropriate.

*For example*, use System.Collections instead of System.Collection. Brand names and acronyms are exceptions to this rule, however. For example, use System.IO instead of System.IOs.

**X DO NOT** use organizational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organize the hierarchy of namespaces around groups of related technologies.

**X DO NOT** use the same name for a namespace and a type in that namespace.

*For example*, do not use Debug as a namespace name and then also provide a class named Debug in the same namespace. Several compilers require such types to be fully qualified.

## Names of Classes, Structs, and Interfaces

**✓ DO** name classes and structs with nouns or noun phrases, using PascalCasing. This distinguishes type names from methods, which are named with verb phrases.

**✓ DO** name interfaces with adjective phrases, or occasionally with nouns or noun phrases.

**✓ DO** prefix interface names with the letter I, to indicate that the type is an interfaceI; for example. For example, IComponent (descriptive noun), ICustomAttributeProvider (noun phrase), and IPersistable (adjective) are appropriate interface names. As with other type names, avoid abbreviations.

**X DO NOT** give class names a prefix (e.g., "C").

**✓ CONSIDER** ending the name of derived classes with the name of the base class. This is very readable and explains the relationship clearly. Some examples of this in code are: ArgumentOutOfRangeException, which is a kind of Exception, and SerializableAttribute, which is a kind of Attribute. This naturally doesn’t work in all cases; for example, a button is a type of control although Control doesn’t appear in its name.

## Names of Generic Type Parameters

**✓ DO** name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value.

**✓ CONSIDER** using T as the type parameter name for types with one single-letter type parameter.

public int IComparer<T> { ... }

public delegate bool Predicate<T>(T item);

public struct Nullable<T> where T : struct { ... }

**✓ DO** prefix descriptive type parameter names with T.

public interface ISessionChannel<TSession> where TSession : ISession

{

TSession Session { get; }

}

**✓ CONSIDER** indicating constraints placed on a type parameter in the name of the parameter. For example, a parameter constrained to ISession might be called TSession.

## Names of Type Members

Types are made of members: methods, properties, events, constructors, and fields. The following sections describe guidelines for naming type members.

### Names of Methods

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

**✓ DO** give methods names that are verbs or verb phrases.

public class String

{

public int CompareTo(...);

public string[] Split(...);

public string Trim();

}

### Names of Properties

**✓ DO** name properties using a noun, noun phrase, or adjective.

**✓ DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."

**✓ DO** name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.

**X DO NOT** have properties that match the name of "Get" methods as in the following example:

public string TextWriter { get {...} set {...} }

public string **Get**TextWriter(int value) { ... }

### Names of Fields

**✓ DO** use PascalCasing in field names.

**✓ DO** name fields using a noun, noun phrase, or adjective.

**X DO NOT** use a prefix for field names known as Hungarian Notation.

**X DO NOT** include the parent class name within a property name.

Should be: Customer.Name **NOT** Customer.CustomerName

## Naming Parameters

**✓ DO** use camelCasing in parameter names.

**✓ DO** use descriptive parameter names.

**✓ CONSIDER** using names based on a parameter’s meaning rather than the parameter’s type.

## Coding Style

|  |  |
| --- | --- |
| **Code** | **Style** |
| Source Files | One Namespace per file and one class per file. |
| Curly Braces | On new line. Always use braces when optional. |
| Indention | Use tabs with size of 4. |
| Comments | Use // or /// but not /\* … \*/ and do not flowerbox. |
| Variables | One variable per declaration. |

## 

## Language Usage

|  |  |
| --- | --- |
| **Code** | **Style** |
| Native Data Types | Use built-in C# native data types vs .NET CTS types. |
|  | (Use int NOT Int32) |
| Enums | Avoid changing default type. |
| Generics | Prefer Generic Types over standard or strong-typed classes. |
| Properties | Never prefix with Get or Set. |
| Methods | Use a maximum of 7 parameters. |
| Ternary conditions | Avoid complex conditions. |
| foreach statements | Do not modify enumerated items within a foreach statement. |
| Conditionals | Avoid evaluating Boolean conditions against true or false. |
|  | No embedded assignment. |
|  | Avoid embedded method invocation. |
| Exceptions | Do not use exceptions for flow control. |
|  | Use throw; not throw e; when re-throwing. |
|  | Only catch what you can handle. |
|  | Use validation to avoid exceptions. |
|  | Derive from Execption not ApplicationException. |
| Events | Always check for null before invoking. |
| Dispose() & Close() | Always invoke them if offered, declare where needed. |
| AssemblyVersion | Increment manually. |

# Writing clean code

## Poor Names

### Mysterious Names

Here are a few examples of Mysterious Names, where the identifiers must be looked through the code to understand what they are representing.

SqlDataReader dr1;

int od;

void Button1\_Click();

class Page1 {};

var m\_objCollection = new List<string>();

The code should be clean and not force the user to look elsewhere to understand what is occurring. Here are some examples of non-mysterious names:

SqlDataReader reader;

int overdueDays;

void CheckAvailability\_Click();

class ViewCustomerPage { }

var countryNames = new List<string>();

### Meaningless Names

This is when the names of the methods are vague and force the developer to look at the implementation to see what is occurring.

void BeginCehckFunctionality\_StoreClientsSideCheckboxIDsArray();

This is usually due to the method is doing *too many things*. The general rule of thumb is that a method should contain 10 lines of code. A method should only being doing 1 thing and doing it well. When the methods are kept small, coming up with a meaningful name is much easier.

### Ambiguous Names

bool MultiSelect() { }

This is a situation where it’s is unclear on what the method is doing. Is this name telling me “*if multiple items are selected*” or is it “*to select multiple items*”. The implementation must be looked at to see what the method is or will doing

### Noisy Names

Customer theCustomer;

List<Customer> listOfApprovedCustomers;

The additional verbiage doesn’t add any value to the names where the following are shorter and more direct, less noise:

Customer customer;

List<Customer> = approvedCustomers;

## Poor Method Signatures

Here is an example

void Parse(int command);

The issue is that Parse usually means to get a string and converting it to a different object. For an example, there could be a Parse object that gets a string representation of a date time and return a date object. So the method above should be more like

int Parse(string command);

Another example would be:

var user = userService.GetUser("username", "password", true);

var user = userService.GetUser("username", null, false);

Where the implementation of GetUser would need to be looked at to determine what the code is doing. By separating the Boolean function into 2 methods, a rewrite could be something like:

var user = userService.Login("username", "password");

var user = userService.GetUser("username");

Now, it’s clearer that the first method will log the user in and the 2nd method will return a user.

## Long Parameter List

According to Microsoft C# team, the number of parameters should not exceed 7 but the more industry standard is 5. This example is assuming that there are several methods with similar parameters.

GetAmlStatusResults(DateTime dateFrom, DateTime dateTo, User user, int locationId, LocationType locationType, int? formId = null);

One thing that could help shorten the parameter list would be to introduce, a DateRange nested class.

public class DateRange

{

private DateTime dateFrom;

private DateTime dateTo;

}

This is assuming there are other methods that take the DateFrom/To, user, locationId, LoactionType a Top-Level class could be created to hold these variables and the final result would look like:

GetAmlStatusResults(Query query, int? formId = null);

General rule is that anything more than 5 parameters could be a potential code smell and it is necessary to remove unnecessary parameters or encapsulate the logically related ones into its own class.

## Output Parameters

int count = 0;

public IEnumerable<Customer> GetCustomers(int pageIndex, out totalCount) { }

This method returns a list of customer on that page and also returns the total number of customers in the db. The issue is that, we would need to declare and initialize a variable and pass it as an out to the method. Data going in and coming out through the arguments seems unnatural.

Here a Tuple could be used and the method would be rewritten as

public Tuple<IEnumerable<Customer>, int> GetCustomers(int pageIndex) { }

Another way would be to create a class, to return the values

public class GetCustomerResult

{

public IEnumerable<GetCustomerResult> Customers { get; set; }

public int TotalCount { get; set; }

}

public GetCustomerResult GetCustomers(int pageIndex) { }

## Inline Variables

Variable declarations should be where the variables are needed and not all crammed at the top of the method. At one time, the compiler needed all variables to be declared but this was back in the 90’s. Declaring the variables when needed makes the code easier to follow and less scrolling to see what and where the variable may or may not have been initialized.

There is also no need to assign an initial value to a variable, in cases like this:

int customerCount = 0;

customerCount = customer.Grid \* customer.Pages;

The declaration and assignment should be joined

var customerCount = customer.Grid \* customer.Pages;

Naturally, this can only be done if the variable isn’t set by a condition. Here is an example where the variable must be initialized.

int foundationCount = 0;

if(accountType == 'Foundation')

foundationCount = 1;

## Magic Numbers

if(status == 1)

{

...

} else if (status == 2)

{

...

}

Magic numbers are numbers that have no meaning to the developer. What does status 1 or 2 mean? These make the code hard to read, hard to understand, and hard to change. These should be avoided at all times. The best case is to make an enum.

public enum DocumentStatus

{

Draft = 1,

Lodged = 2

}

if(status == DocumentStatus.Draft)

{

. . .

} else if(status == DocumentStatus.Lodged)

{

. . .

}

Now throughout the code, the DocumentStatus can be referenced without any Magic Numbers that can lead to errors.

## Nested Conditionals

### Ternary Operator

if(a)

c = someValue;

else

c = anotherValue;

This can be rewritten as

c = (a) ? someValue : anotherValue;

Another example would be:

discount = (customer.TotalOrders > 50) ? 0.1f : 0.01f;

General rule is NOT to use more than 1 Ternary Operator per expression. It makes it much too difficult to understand. For example,

c = a ? b : d ? e : f;

### Simplify true/false

if(customer.TotalOrders > 50)

isGoldCustomer = true;

else

isGoldCustomer = false;

This can be simplified by:

isGoldcustomer = customer.TotalOrders > 50;

Another example would be:

public bool isTrust

{

get { return AccountOwnership == "trustIndi" }

}

Where true will be returned if AccountOwnership is equal to trustIndi

### Early Exit

if(a)

{

if(b)

{

statement

}

}

Sometimes nested if statements can be hard to follow and a much similar approach would be something like:

if(!a)

return;

if(!b)

return;

statement

An even shorter way to write this would be:

if(!a || !b)

return;

statement

### Swap Order

if(a)

{

if(b)

{

isValid = true;

}

}

if(c)

{

if(b)

{

isValid = true;

}

}

Looking at the above scenario, we can see isValid is true only when b is true. The above code could be rewritten like this, which is much easier to understand.

if(b && (a || c))

{

isValid = true;

}

In reality, this could also be shortened again by the Simplify true/false example above, to

isValid = (b && (a || c));

These methods can be used only, if it makes since. Once the complexity is greater than the readability, you have gone overboard and need to reverse course. Something like this is way too much:

isValid = (a && (b || C) && !d || e && (f && !g || h))

### Switch Statements

Switch statement are faster to execute than the if-else-if Statements. This is true because the compiler can generate a jump table. Where a fixed block of memory is reserved contains a jump instruction to the location where the actual code for the *case* resides. Bottom line, the longer the list, the better a switch statement is over a series of if statements.

### Long Methods

Anything over 10 lines of code, it *may be* doing too many things. Methods should specialize in one thing only and should do it well. Long methods are:

* Hard to understand
* Hard to change
* Hard to re-use

Methods like classes should be cohesive.

# Quick Summary

Consistency is the key to maintainable code. This statement is most true for naming your projects, source files, and identifiers including Fields, Variables, Properties, Methods, Parameters, Classes, Interfaces, and Namespaces.

## Naming Guidelines

* Always use Camel Case or Pascal Case names no Hungarian Notation.
* Avoid ALL CAPS and all lowercase names. Single lowercase words or letters are acceptable.
* Do not use names that begin with a numeric character.
* Always choose meaningful and specific names.
* Avoid using abbreviations unless the full name is excessive.
  + Avoid abbreviations longer than 5 characters.
  + Any Abbreviations must be widely known and accepted.
  + Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.
* Do not include the parent class name within a property name.  
  Example: Customer.Name NOT Customer.CustomerName
* Try to prefix Boolean variables and properties with “Can”, “ Is” or “ Has”.
* Methods should use Pascal Case and be Verbs or verb phrases  
  Example: public void Execute() {…}
* Property should use Pascal Case (noun, noun phrase, or adjective) and represent the entity it returns  
  public string Name

{

get {…}

set {…}

* Field variables should use Pascal Case (noun, noun phrase, or adjective)
* Parameter variables should use Camel Case
* Constants or Static should use Pascal Case
* Acronyms should not be all in upper case, the first character can be upper where the rest are lower case:
  + Use Http vs HTTP (ie: HttpContext)
  + Use Id vs ID (ie: CustomerId)
  + User Aml vs AML (ie: AmlType)

## Coding Style

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are key to creating maintainable code. The following sections describe the preferred way to implement C# source code in order to create readable, clear, and consistent code that is easy to understand and maintain.

* All comments should be written in the same language, be grammatically correct, and contain appropriate punctuation.
* Use // or /// but never /\* … \*/
* Use inline-comments to explain assumptions, known issues, and algorithm insights.
* Do not use inline-comments to explain obvious code. **Well written code is self-documenting**.

## Variables & Types

* Try to initialize variables where you declare them.
* Always choose the simplest data type, list, or object required.
* Always use the built-in C# data type aliases, not the .NET common type system (CTS).

Example:

short NOT System.Int16 int NOT System.Int32

long NOT System.Int64

* Only declare member variables as private. Use properties to provide access to them with public, protected, or internal access modifiers.
* Only use long for variables potentially containing values too large for an int.
* Try to use double for fractional numbers to ensure decimal precision in calculations.
* Only use float for fractional numbers that will not fit double or decimal.
* Try to use decimal when fractional numbers must be rounded to a fixed precision for calculations. Typically this will involve money.
* Avoid using inline numeric literals (magic numbers). Instead, use a Constant or Enum.
* Declare readonly or static readonly variables instead of constants for complex types.
* Only declare constants for simple types.
* Avoid boxing and unboxing value types.  
  Example:

int count = 1;

object refCount = count; // Implicitly boxed.

int newCount = (int)refCount; // Explicitly unboxed.

## 4.6 Object Composition

* Avoid declaring methods with more than 5 parameters. Consider refactoring this code.
* Try to replace large parameter-sets (> than 5 parameters) with one or more class or struct parameters – especially when used in multiple method signatures.

# C# remove code smells

This section will highlight a few code smells and improvements.

## Null-Conditional Operator

### ? Operator

**Messy**:

string result = value;

if (value != null) // Skip empty string check for elucidation

{

result = value.Substring(0, Math.Min(value.Length, length));

}

return result;

**Cleaner with ?.:**

return value?. Substring(0, Math.Min(value.Length, length));

If the value of the object is null, the null-conditional operator will return null. Note the question mark prior to the dot operator (?.). So either null or the value will be returned.

The ? can also work on indexes:

Customer first = customers?.[0];

// returns null if customers is null or the index if not null

### ?? Operator

**Messy**:

if (customerAge == null)

actualAge = -1;

else

actualAge = customerAge;

**Cleaner with ??**

int actualAge = customerAge ?? -1;

Where the customer’s age will be returned, if not null. If null, then the default value of -1 will be returned.

The 2 Null-conditional operators can be combined:

return person?. FirstName ?? "Unspecified";

Where if person is null, null is returned, if not then the FirstName will be returned if it’s not null, if null the "Unspecified" will be returned.

### String Interpolation

**Messy**:

return string.Format("{0} {1}", FirstName, LastName);

**Cleaner with $**:

return $"{FirstName} {LastName}";

Another example:

public string GetFormattedGradePoint() => $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Average()}";

## Exception Filters

Exception Filters are clauses that determine when a given catch clause should be applied. If the expression used for an exception filter evaluates to true, the catch clause performs its normal processing on an exception. If the expression evaluates to false, then the catch clause is skipped.

### When

**Messy**:

catch (System.Net.Http.HttpRequestException e)

{

if (e.Message.Contains("301"))

return "Site Moved";

else

throw;

}

**Cleaner with when**:

catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))

{

return "Site Moved";

}

### Throw Expressions

This feature enables using throw expressions in initialization expressions  
**Messy**:

loadedConfig = LoadConfigResourceOrDefault();  
if (loadedConfig == null)  
 throw new InvalidOperationException("Could not load config");

**Cleaner**

private ConfigResource loadedConfig = LoadConfigResourceOrDefault() ??   
 throw new InvalidOperationException("Could not load config");

# References

“MSDN: .NET Framework Developer’s Guide: Common Type System”, Microsoft Corporation, https://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.110).aspx

“MSDN: C# Language Specification”, Microsoft Corporation, 2017, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification

“MSDN: Design Guidelines for Class Library Developers”, Microsoft Corporation, https://msdn.microsoft.com/en-us/library/ms229042(v=vs.110).aspx

“MSDN: Exception handling guidelines”, Eric Gunnerson, Microsoft Corporation, https://blogs.msdn.microsoft.com/ericgu/2006/02/15/exception-handling-guidelines/

“Which type should I use in C# to represent numbers?”, Luca Bolognese, Microsoft Corporation, <https://blogs.msdn.microsoft.com/lucabol/2007/02/27/which-type-should-i-use-in-c-to-represent-numbers/>

“MSDN: Capitalization Conventions”, Microsoft Corporation,

<https://msdn.microsoft.com/en-us/library/ms229043(v=vs.110).aspx>

“Framework Design Guidelines”, Kryzysztof Cwalina and Brad Abrams,

http://accorsi.net/docs/Framework%20Design%20Guidelines\_%20Conventions%20-%20Krzysztof%20Cwalina.pdf