|  |  |
| --- | --- |
| https://cdnp-2f3a.kxcdn.com/blog/wp-content/uploads/2015/04/15-Most-Popular-Programming-Languages-You-Must-Learn-in-2015.png  c360  CODING  guidelines    **Good to know**  On this document, you will find all the technical information needed for work property in the C360 deparment. This document is truly open to modifications, so we invite you to add or modify as content as you want.  Enjoy!  http://powiat-zielonogorski.pl/var/powiat/storage/images/powiat/aktualnosci/150-lecie-nestle/25509-1-pol-PL/150-lecie-Nestle_imageextralarge.jpg | **Best practices**  **────**    **Patterns and principles**  **────**    **Usage guidelines**  **────**    **Testing**  **────**    **Backend and frontend environments**    **DSU GLOBAL HUB**  **Barcelona   ────**  ***Customer 360***  v.1.0  June 2017 |

Contents

[Capitalization conventions 4](#_Toc486334606)

[Casing Styles 4](#_Toc486334607)

[Pascal case 4](#_Toc486334608)

[Camel case 4](#_Toc486334609)

[Uppercase 4](#_Toc486334610)

[Casing by language 4](#_Toc486334611)

[.Net 4](#_Toc486334612)

[JavaScript 5](#_Toc486334613)

[General conventions 5](#_Toc486334614)

[Naming conventions 5](#_Toc486334615)

[Assemblies and Namespaces 5](#_Toc486334616)

[Considerations regarding namespaces 6](#_Toc486334617)

[Definition of Classes, Interfaces and Enumerations 6](#_Toc486334618)

[Enumerations 6](#_Toc486334619)

[.Net considerations 6](#_Toc486334620)

[JavaScript considerations 7](#_Toc486334621)

[Methods and properties definitions 7](#_Toc486334622)

[.Net considerations 7](#_Toc486334623)

[JavaScript considerations 8](#_Toc486334624)

[Security 8](#_Toc486334625)

[Patterns and principles 8](#_Toc486334626)

[About patterns 8](#_Toc486334627)

[SOLID principles 9](#_Toc486334628)

[DRY principle 11](#_Toc486334629)

[KISS principle 11](#_Toc486334630)

[The Boy Scout rule 12](#_Toc486334631)

[Testing 13](#_Toc486334632)

[About User Interface testing 13](#_Toc486334633)

[About Usability Testing 13](#_Toc486334634)

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Changed by** | **Description** |
| 1.0 | 26/06/2017 | Iban Balasch | Initial document |
|  | 13/11/2017 | Iban Balasch | Application Insights issues |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Capitalization conventions

## Casing Styles

We define only three different casing styles:

### Pascal case

In the Pascal case (or upper camel case), the first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. Example: *BackColor*

### Camel case

In the Camel case (or lower camel case), the first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. Example: *backColor*

### Uppercase

All letters in the identifier are capitalized. Example: *BACKCOLOR*

## Casing by language

### .Net

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Case** | **Example** |
| Class | Pascal | **ApplicationDomain** |
| Enumeration type | Pascal | **ErrorLevel** |
| Enumeration values | Pascal | **FatalError** |
| Event | Pascal | **ValueChanged** |
| Exception class | Pascal | **WebException** |
| Read-only static field | Pascal | **RedValue** |
| Interface | Pascal | **IDisposable** |
| Method | Pascal | **ToString** |
| Namespace | Pascal | **System.Drawing** |
| Parameter | Camel | **typeName** |
| Property | Pascal | **BackColor** |

### JavaScript

In this language, the rule to follow is very simple: everything must be Camel case, the Uppercase must be used only for environment constants.

# General conventions

## Naming conventions

The general naming conventions discuss choosing the best names for the elements in your libraries. The name that we choose for any object must follow these concepts/principles:

* Easily readable.
  + The name must provide as much information as possible related to it, to avoid any kind of confusion.
  + “ScrollHorizontally” is a better name than “ScrollX”
* A long name is better than using abbreviations or acronyms.
  + “UseEnvironment” is better than “UseEnv”
* Do not use underscores (**see note below\***), hyphens, or any other nonalphanumeric characters.
  + “myProperty” instead “\_property”.
  + Exception: the underscores must be used in the global private properties of a class.
* Avoid using identifiers that conflict with keywords of widely used programming languages.
  + Don’t use names like “Exception” which could and will create confusion.
* All names must start with a letter

(\*) Exception to this rule: In .Net, the private global properties of a class must use it.

## Assemblies and Namespaces

The assemblies and namespaces must follow a list of rules to assign them a valid name. These rules are:

* For Assemblies, the pattern to follow is ***<****Company****>.<****Department****>.<****Component****>.dll***
  + Example: Nestle.C360.Business.dll
* For Namespaces, the pattern is ***<****Company****>.<****Department****>****.****<****Product****>****.****<****Technology****>****.****<****Feature****>***.
  + Example: Nestle.C360.Business.Services.Customer

### Considerations regarding namespaces

In addition, the namespaces must follow some guidelines in their nomenclature, basically to avoid clashes or conflicts with another generic type. Remember these points:

* Don’t use generic namespaces that use your current framework and/or library.
* Don’t introduce generic type names (Linq, Node, Message, etc.) in the definition. There is a very high probability it would lead to type name conflicts in common scenarios.
* The company prefix can help to differentiate from other companies that use a similar namespace.
* Use Pascal casing (both for namespaces and assemblies too).
* Use plural namespaces only if it’s appropriate.
* Don’t use the same name for a namespace and a class inside of it. This causes confusion and duplicates the concept.

## Definition of Classes, Interfaces and Enumerations

The following guidelines apply to the naming conventions for all the above:

* Don’t use prefixes in a class name with a single character, only the interface’s name can do this.
  + CMyClass is not allowed
  + IMyClass is allowed
* If a class is derived from a framework class, its name must end with that name.
  + If we create MyClass based on the Exception class, the final name must be MyClassException.
  + If our class inherits from another class (from our environment, not from a framework class), then use this rule at your discretion.
* Whenever possible, try to make sure that the interface and the class that implements it have the same name.

### Enumerations

This object has some peculiarities, namely:

* Don’t use a prefix or suffix for an enumeration type
* Use a singular name (unless the description of the enumeration type is a plural concept)
* If it’s possible, add a Description attribute to have a better chance of retrieving information from it.

### .Net considerations

For .Net environments, we must follow these rules:

* Include the summary information section for each class.
* A class with just one generic type parameter will have one single-letter description
  + MyClass<T>
* If the generic type parameter has more than one parameter, we must provide a descriptive suffix for each one.
  + MyClass<TKey, TValue>
* Add the suffix Attribute to a custom attribute class
* Add the suffix EventHandler to types that are used in events
* Add the suffix Callback to the name of a delegate that is not an event handler
* Don’t add the suffix Delegate to a delegate
* Add the suffix EventArgs to any classes that extend the System.EventArgs

### JavaScript considerations

For JavaScript environment, the rules to follow are:

* Use prototype to extend functionality.
* In ES6, the class’s name must be Pascal Case.
* In ES6, use the object’s constructor for each object.

## Methods and properties definitions

The naming convention for methods are, basically, the following:

* Give methods names that are verbs or verb phrases. The method’s name must be self-describing.
  + Wrong: GetSome (int id)
  + Right: GetObjectTypeNameById (int identifier)
* Include the summary information for each public method. Private methods can be documented at your discretion.

Properties must follow a single rule: they must be self-descriptive and it must be easy to understand the purpose and type with just its definition.

### .Net considerations

For .Net environments, we must follow these rules (including classes, properties and variables):

* Use, wherever possible, the interface of a base type instead of the implementation class.
  + Correct: Dictionary myDictionary = new Dictionary<string, string> ()
  + Nice to have: IDictionary myDictionary = new Dictionary<string, string> ()
* Avoid using ***var***. Use the specific type instead.
  + We always prefer a strong specific type code.
* If an object implements a base object from System.Collections (like IDictionary or IList), we will add a suffix with its base type (example: MyObjectList)

### JavaScript considerations

For JavaScript environments, the rules to follow are:

* Use camelCase for methods and identifiers names
* Global variables and constants will use UPPERCASE
  + Try to avoid global variables because they can be overwritten by other scripts
* Don’t use underscores for method and/or property names.
* All names must start with a letter
* Always put spaces around operators (= + - \* /) and after commas
* Always use four (**4**) spaces for indentation of code blocks
* Don’t use tabs (tabulators) for indentation.
  + Customize your editor indent by spaces instead tabs
* Always end a statement with a semicolon
* Put the opening bracket on the same line as the object name
* Place the closing bracket on a new line, without leading spaces
* Always end an object definition with a semicolon
* For readability, avoid lines longer than **120** characters
  + Customize your editor to do that
* Use lowercase for file names.
  + Most web servers are case sensitive about file names, “Barcelona.png” is not the same as “barcelona.png”
* The code in both development and test environments must be without minification or encryption / obfuscation.
* The code in production environments must be minified and encrypted.

# Security

The thing-to-know about security is: first, the security must be compliant by design. Then must be compliant with the C360 department and, finally, must be compliant with the current platform.

# Patterns and principles

## About patterns

Some of the most important to take into consideration, in .Net and JavaScript environments, are:

* Factory
* Prototype
* Singleton
* Adapter
* Decorator
* Composite
* Observer
* Chain of responsibility

Use software design patterns provides a lot of benefits:

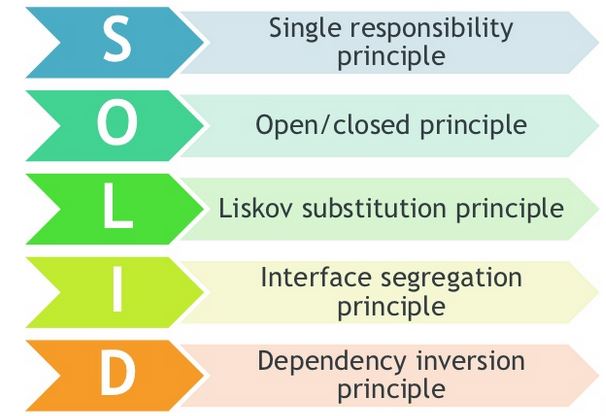
* Give a selection of tried and tested solution to work with
* Can be applied to any language that supports object-orientation (yes, we include JavaScript!)
* Well documented
* Save effort in the implementation stage
* If the solution is easier to comprehend, then will also be easier to maintain.

Remember: we encourage and demand that they must be used (in certain conditions, remember that uses a lot of patterns is an anti-pattern).

More info at: <https://en.wikipedia.org/wiki/Software_design_pattern>

## SOLID principles

These are the first, most common and important principles and usually the first to be forgotten. It is extremely important to follow them to arrive at maintainable, clean and easy to understand code.



Right now, we will talk only about the **S** because is the first and more important principle to follow.

The **S** of solid talks about single responsibility. This means that a class has only a single responsibility. We must follow this predicate to the method itself, if a method performs different actions instead of a unique action, we must refactor it to remove this mistake and provide a better class to put under tests.

Let’s see it in an example:



The desired result must isolate the code as much as possible (**\***) to follow the Single Responsibility principle:



(**\***) For example, in the code above, the inner *foreach* can be isolated too.

More information at: <https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)>

## DRY principle



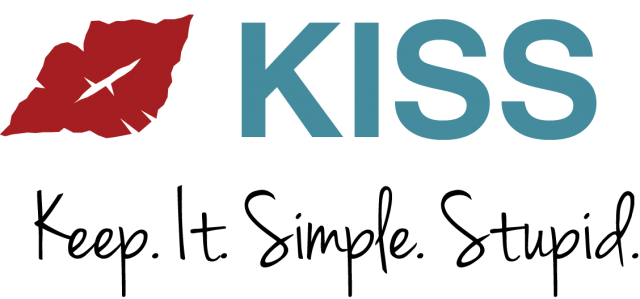
The DRY principle has the next statements as keys:

* Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
* DRY code means all uses of data change simultaneously rather than needed to be replicated
* DRY is the fundamental reason to use variables and functions in all programming
* Bugs love a wet mess.

So, as a summary, if you see a piece of code that is used twice in the application, follow [the boy scout rule](#_The_Boy_Scout) and refactor it!

More information at: <https://en.wikipedia.org/wiki/Don%27t_repeat_yourself>

## KISS principle



That’s it. Let’s follow this simple principle, it’s basically to follow the rule about simplification and readability is better than complex and unmaintainable code.

An example of this principle is to avoid this code:



Try to replace it with something like this:



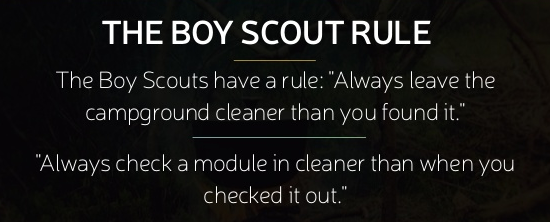
The reason to replace the above code is because it is more readable, we add more validation steps, it provides a better debug process and the code generated can be isolated if needed.

More information at: <https://en.wikipedia.org/wiki/KISS_principle>

## The Boy Scout rule

Basically, this rule means that if you see something that is wrong in the code that you are working on, something that needs a refactor: **fix it**! This must be a part of your duties.

Remember:



More information at: <http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule>

# About logs

On the .Net Framework environments, we suggest use log4net (<https://logging.apache.org/log4net>) to trace logs.

In .Net Core we have a log embedded in its own framework, so use this instead log4net.

In JavaScript, if you want to show something in console, create a wrapper

For all the environments, we need to activate Azure Applications Insights for trace collection, on this case, you will need to registry your application in a new entry and, if you want to generate trace, you have this commands that allow you:



# Testing

For an effective test planning, we must follow the next statements:

* Identify the requirements to be tested.
* Identify which test must be done for each module
* Review and verify that the test data used are the adequate
* Identify the expected result for each test
* Document the expected test result (commonly on Jira)
* Successful unit testing is required before the unit is eligible for component integration testing
* Unsuccessful testing requires a problem report to be generated. On that document, shall describe the test case, the problem encountered, its possible cause and the sequence of events that led to the problem.

## About User Interface testing

The User Interface (UI) has its own test planning that differs than the others. For an effective UI test planning, we must cover these issues:

* Check that the UI is fully compatible with Internet Explorer 11 (corporative browser, this rule is a must-to-have)
* Check for broken links
* Test the functionality of all controls (buttons, check boxes, radio buttons, …)
* Test the loading time
* Test the asynchronous loading objects
* Test the functionality about the objects on screen have the expected result on each action that they accept.
* Try website responsive and readable for all screen resolutions (right now, we will not perform this tests)

## About Usability Testing

Usability testing is the gold standard by which you can determine if the design of an application meets the needs of its intended users and allows them to work productively.

But, right now, we will not perform any kind of usability testing for our applications.