



monoTM

Coding Guidelines

A Conversion Document from mono-project Coding Guidelines.

Monkey: Paulino Padial López

Index

Coding Guidelines.....	1
Style Guidelines	3
Indentation	3
Performance and Readability	3
Where to put spaces	3
Where to put braces	4
Use whitespace for clarity	5
File headers	5
Casing	6
Line length and alignment	6
RCS and CVS tags	7
ChangeLogs	7
File formats	7
Warnings	8
Conditional compilation	8
Missing Implementation Bits	8
NotImplementedException	9
Examples	9
Best Practices	10
Tests	11
Check all arguments	11
Be careful about freeing memory	11
Watch for integer [under over]flows	11
Locking and Threading	11
Deadlocks	11
Don't lock on Types and Strings	13

Style Guidelines

In order to keep the code consistent, please use the following conventions. From here on `good` and `bad` are used to attribute things that would make the coding style match, or not match. It is not a judgement call on your coding abilities, but more of a style and look call. Please try to follow these guidelines to ensure prettiness.

Indentation

Use 8 space tabs for writing your code (hopefully we can keep this consistent). If you are modifying someone else's code, try to keep the coding style similar.

Since we are using 8-space tabs, you might want to consider the Linus Torvalds trick to reduce code nesting. Many times in a loop, you will find yourself doing a test, and if the test is true, you will nest. Many times this can be changed.

Example:

```
for (i = 0; i < 10; i++) {  
    if (something (i)) {  
        do_more ();  
    }  
}
```

This take precious space, instead write it like this:

```
for (i = 0; i < 10; i++) {  
    if (!something (i))  
        continue;  
    do_more ();  
}
```

Switch statements have the case at the same indentation as the switch:

```
switch (x) {  
case 'a':  
    ...  
case 'b':  
    ...  
}
```

Performance and Readability

It is more important to be correct than to be fast.

It is more important to be maintainable than to be fast.

Fast code that is difficult to maintain is likely going to be looked down upon.

Where to put spaces

Use a space before an opening parenthesis when calling functions, or indexing, like this:

```
method (a);  
b [10];
```

Do not put a space after the opening parenthesis and the closing one, ie:

good:

```
method (a);    array [10];
```

bad:

```
method ( a );    array[ 10 ];
```

Where to put braces

Inside a code block, put the opening brace on the same line as the statement:

good:

```
if (a) {
    code ();
    code ();
}
```

bad:

```
if (a)
{
    code ();
    code ();
}
```

Avoid using unnecessary open/close braces, vertical space is usually limited:

good:

```
if (a)
    code ();
```

bad:

```
if (a) {
    code ();
}
```

When defining a method, use the C style for brace placement, that means, use a new line for the brace, like this:

good:

```
void Method ()
{
}
```

bad:

```
void Method () {
}
```

Properties and indexers are an exception, keep the brace on the same line as the property declaration.

Rationale: this makes it visually simple to distinguish them.

good:

```
int Property {
    get {
        return value;
    }
}
```

bad:

```
int Property
{
    get {
        return value;
    }
}
```

Notice how the accessor "get" also keeps its brace on the same line.

For very small properties, you can compress things:

ok:

```
int Property {
    get { return value; }
    set { x = value; }
}
```

Empty methods: They should have the body of code using two lines, in consistency with the rest:

good:

```
void EmptyMethod ()
{
}
```

bad:

```
void EmptyMethod () {}

void EmptyMethod ()
{ }
```

Use whitespace for clarity

Use white space in expressions liberally, except in the presence of parenthesis:

good:

```
if (a + 5 > method (blah () + 4))
```

bad:

```
if (a+5>method(blah()+4))
```

File headers

For any new files, please use a descriptive introduction, like this:

```
//  
// System.Comment.cs: Handles comments in System files.  
//  
// Author:  
//   Juan Perez (juan@address.com)  
//  
// Copyright (C) 2002 Address, Inc (http://www.address.com)  
//
```

If you are modifying someone else's code, and your contribution is significant, please add yourself to the Authors list.

Casing

Argument names should use the camel casing for identifiers, like this:

good:

```
void Method (string myArgument)
```

bad:

```
void Method (string lpstrArgument)

void Method (string my_string)
```

Instance fields should use underline as a separator:

good:

```
class X {
    string my_string;
    int    very_important_value;
}
```

bad:

```
class X {

    string myString;
    int    veryImportantValue;
}
```

worse:

```
class X {
    int    _intval;
    string m_string;
}
```

The use of "m_" and "_" as prefixes for instance members is highly discouraged.

An exception to this rule is serializable classes. In this case, if we desire to have our serialized data be compatible with Microsoft's, we must use the same field name.

Line length and alignment

Line length: The line length for C# source code is 80 columns.

If your function declaration arguments go beyond this point, please align your arguments to match the opening brace, like this:

```
void Function (int arg, string argb,  
              int argc)  
{  
}
```

When invoking functions, the rule is different, the arguments are not aligned with the previous argument, instead they begin at the tabbed position, like this:

```
void M ()  
{  
    MethodCall ("Very long string that will force",  
               "Next argument on the 8-tab pos",  
               "Just like this one");  
}
```

RCS and CVS tags

Some users like to use the special RCS/CVS tags in their source code: \$id\$, \$log\$ and so on.

The use of these is not permitted on the Mono source code repository. This metadata belongs on a ChangeLog or in the SVN metadata.

ChangeLogs

ChangeLogs are the files that we use to track the project history. ChangeLogs are found one per directory, or in small projects, one per project.

The format looks like this:

```
2004-11-19 Raja R Harinath <rharinath@novell.com>

* Makefile (%-profiles): Go through an intermediate
set of rules. Move body to ...
(profiles-do--%): ... this.
(profiles-do--run-test): Customized rule that usefully
runs with 'make -j' and 'make -k'.
(profiles-do--all, profile-do--%--all): Orchestrate
the bootstrap process.

* file.cs (MainForm): Updated version.
```

The date, author, email address in the first line.

From that point on a list of changes in a file-by-file basis, describing what changes were done.

This information must be cut and pasted into your commit message, so the information ends up in two places: in the subversion repository metadata and also on the source code distribution (which does not have the Subversion metadata).

File formats

Historically our repository has used a mix of line-endings, this is a mistake that we are trying hard to fix.

For existing files, please make sure that you do not convert the file, as that causes us to lose precious history (the full file is committed).

For new files that you create, please make sure that you use Subversion's support for mapping the line endings automatically, after adding your file:

```
$ svn add file.cs
```

Execute this command:

```
$ svn propset svn:eol-style native file.cs
```

Which will make the file automatically receive the proper treatment from that point on.

Please verify before committing that your changes won't lose history, you can do this by running:

```
$ svn diff
```

And examining the output.

Warnings

Avoid committing code with warnings to the repository, the use of #pragmas to disable warnings is strongly discouraged, but can be used on unique cases. Please justify the use of the warning ignore clause on a comment.

Do not commit changes to the Makefiles that removes warnings, if anything warnings should be eliminated one at a time, and if not possible, they must be flagged.

Conditional compilation

Ideally we would not need conditional compilation, and the use of `#ifdef` is strongly discouraged. But due to our support for old C# 1.0 compilers we have to use it in a few places.

Try to avoid negative tests that have an else clause, for example:

```
#if !NET_2_0
    CODE_FOR_1_0
#else
    CODE_FOR_2_0
#endif
```

Instead use:

```
#if NET_2_0
    CODE_FOR_2_0
#else
    CODE_FOR_1_0
#endif
```

When a major feature differs across compilation targets, try to factor out the code into a separate class, a helper class or a separate file, and include that in your profile while surrounding that helper file/class with the `ifdefs` to reduce the amount of `ifdefs` in the code.

For instance, this is used for some parts of Grasshopper where the code is `ifdefed` out, when large parts of a file would have been `ifdefed` out, we moved the code into a `MyOtherFile.jvm.cs`

For 2.0 classes, this is even simpler as code can be trivially factored out into

```
MyHelperClass.cli.cs
MyHelperClass.jvm.cs
```

By using partial classes.

Missing Implementation Bits

There are a number of attributes that can be used in the class libraries to flag the specific state of an API, class or field.

The following attributes exist:

MonoDocumentationNote, MonoExtension, MonoInternalNote, MonoLimitation and MonoNotSupported.

If you implement a class and you are missing implementation bits, please use the attribute MonoTODO. This attribute can be used to programatically generate our status web pages:

```
[MonoTODO("My Function is not available on Mono")]
int MyFunction ()
{
    throw new NotImplementedException ();
}
```

Ideally, write a human description of the reason why there is a MonoTODO, this will be useful in the future for our automated tools that can assist in developers porting their code.

Do not use MonoTODO attributes for reminding yourself of internal changes that must be done. Use FIXMEs or other kinds of comments in the source code for that purpose, and if the problem requires to be followed up on, [file a bug](#).

NotImplementedException

In the Mono class libraries, if a library is stubbed out, it is customary to insert the following code snippet:

```
throw new NotImplementedException ();
```

The NotImplementedException exception is used by tools like the [Mono Migration Analyzer](#) (Moma) to report potential problems for people porting their applications.

Notice that in certain conditions, throwing a NotImplementedException is a pattern used in base virtual methods to force derived classes to implement the functionality. This can lead to incorrect reports from the Moma tool, because the tool has no way of knowing that in practice the virtual base method will never be called (it would be better to have an abstract class in this case, but we have to be source and binary compatible). In these cases, instead of throwing the exception directly, call a helper routine, so that it becomes invisible to Moma:

```
Exception GetNotImplemented ()
{
    return new NotImplementedException ();
}

void SomeMethod ()
{
    throw GetNotImplemented ();
}
```

Remember the rule: NotImplementedException should be exposed in the toplevel method that is exposed to developers for Moma to properly work.

Examples

```
class X : Y {
    bool Method (int argument_1, int argument_2)
    {
        if (argument_1 == argument_2)
            throw new Exception (Locale.GetText ("They are
equal!"));

        if (argument_1 < argument_2) {
            if (argument_1 * 3 > 4)
                return true;
            else
                return false;
        }

        //
        // This sample helps keep your sanity while using 8-spaces
for tabs
        //
        VeryLongIdentifierWhichTakesManyArguments (
            Argument1, Argument2, Argument3,
            NestedCallHere (
                MoreNested));
    }

    bool MyProperty {
        get {
            return x;
        }

        set {
            x = value;
        }
    }

    void AnotherMethod ()
    {
        if ((a + 5) != 4) {
        }

        while (blah) {
            if (a)
                continue;

            b++;
        }
    }
}
```

Best Practices

Correctness is essential for the Mono class libraries. Because our code is called by many other people, we often must be more pedantic than most people would be when writing code. These items are guidelines of some things you should check for.

The class libraries are grouped together in the assemblies they belong.

Each directory here represents an assembly, and inside each directory we divide the code based on the namespace they implement.

In addition, each assembly directory contains a Test directory that holds the NUnit tests for that assembly.

We use a new build system which is described by various README files in mcs/build

The build process typically builds an assembly, but in some cases it also builds special versions of the assemblies intended to be used\ for testing.

Tests

When fixing a bug, write a test, so we can make sure that we do not re-introduce bugs in the future accidentally.

When writing new code, write tests to exercise the API.

Write tests when you are exploring an API on Windows, you will have to write test cases anyways to understand how the API works. Instead of writing throw away code, write NUnit tests that will test return values, properties and behavior and then write the new implementation using your tests as a baseline.

Code should never be checked into the repository that breaks the build, or breaks any of the existing tests.

Check all arguments

Public functions must check their arguments in exactly the same way as Microsoft's framework. It is important to throw exceptions such as `ArgumentNullException` rather than `NullReferenceException`. Adding test to the [Test Suite](#) is important for these.

Be careful about freeing memory

In general, whenever you create a class that is disposable, you need to use the `using {}` pattern. This way, resources get disposed. Do *not* rely on finalizers to get run. This can lead to performance problems.

Watch for integer [under|over]flows

Integer overflows and underflows are possible (<http://pages.infinit.net/ctech/20050610-0319.html>) in managed code. They can lead to ugly problem if your code deals with unsafe code, internal calls or p/invoke. So for every integer parameter you should ask yourself how your code would react to a `Int32.MaxValue` or a `Int32.MinValue` (substitute `Int32` for your integral type). Unit tests are a nice way to remember that you reviewed your code.

Locking and Threading

As humans, we are trained to think about one thing at a time. We, by nature, are not multi-taskers. Thus, when we write code, we tend to think about it as if it was the only thing happening on the operating system. Sadly, this is not the case. In Mono, we are providing a base framework for others to use. This means that our code can often be called from multiple threads. Our code might be running on 8 cpu server with hyperthreading.

Deadlocks

```
object lockA;
object lockB;

void Foo () {
    lock (lockA) {
        lock (lockB) {
        }
    }
}

void Bar () {
    lock (lockB) {
        lock (lockA) {
        }
    }
}
```

In managed code, the most common way to deadlock is to have reverse locking order:

Imagine that one thread is inside Foo and has just acquired lock A, while another thread is in Bar and just acquired lock B. Now, the Foo thread wants lockB, which is owned by Bar. However, in order for Bar to release it, it must acquire lockA. Nobody can make progress and the program deadlocks. In order to avoid this situation, we **must** ensure that our locks get acquired in the same order each time.

One consequence of this is that we generally can not call code outside of our control under a lock. Imagine this code:

The same kind of deadlock that happened above happens here because the user acquired another lock inside ours. To avoid this, we must not call user code inside our locks.

```
// class library
class Blah {
    Hashtable ht;
    void Foo (int zzz, Entry blah) {
        lock (ht) {
            ht.Add (zzz, blah);
        }
    }

    void Bar ()
    {
        lock (ht) {
            foreach (Entry e in ht)
                EachBar (e);
        }
    }

    virtual void EachBar (Entry e)
    {
    }
}

// User
class MyBlah {
    void DoStuff ()
    {
        lock (this) {
            int i = GetNumber ();
            Entry e = GetEntry ();

            Foo (i, e);
        }
    }

    override void EachBar (Entry e)
    {
        lock (this) {
            DoSomething (e);
        }
    }
}
```

Don't lock on Types and Strings

Code like `lock (typeof (T))` is not good. Type objects can be shared across appdomains, which can lead to unexpected behavior. Also, if the type is public, a user could lock on the type and create deadlocks. Strings are interned, and thus the same type of problems can occur. In general, the best practice is to do:

```
static object lockobj = new object ();  
  
...  
lock (lockobj) {  
    ...  
}
```