Moai Coding Style Guide

Version 1.0
August 21, 2011

Contents

This Guide is Not Very Good

Sorry. We'll eventually capture every special case of our coding style, but we're not there yet. Those of us who have been doing this style for years now can do it consistently and without any ambiguity. It's tougher to distill that knowledge into a comprehensive document, particularly across four languages (C, C++, Objective C and Java). It's tougher yet to organize that knowledge into an orderly and easily digestible fashion. By its nature, a coding style is granular, particular and full of specific, single-purpose rules.

In this document we've done our best with C++ and Objective C, with a slight emphasis on C++ as that is what the majority of our code base is written in.

Use this document to get a feel for some of the more obvious rules, but make a point to look at the existing code base to pick up the nuances. When in doubt ask for clarification.

Religion

There are aesthetic, philosophical and practical reasons for every decision we've made regarding our coding style. None of those matter. What matters is the thousands and thousands of lines of code already written in this style. We are not going to rewrite this code and, most likely, neither are you. For that reason we've decided to omit any kind of explanation, justification, rationalization or argument from this document. Our goal is not to persuade you to join our religion and worship our particular coding style; it is simply to convey what that style is should you choose to use it as a contributor or be required to use it as an employee.

For Contributors

Please consider using our coding style for your contributions to Moai. Of course we will not reject contributions simply because they don't match, but we'll still want to reformat your code before accepting it into the main project. We've found this makes it faster, easier and more pleasant for us to maintain.

For Employees

It is very important to us that our code base remain consistently formatted, so leave your personal coding style at the door when you come to work. Better yet, abandon your personal style altogether and adopt our house style for everything you do. The majority of code you write while working for us will be Moai and you can always unlearn our style when it's time for something new.

A poor strategy is to first write code in your old style and then convert to the house style prior to review. You will only waste time and make mistakes. Learn to get it right the first time, without the need for any revision.

The number one mistake coders make when learning our style is to omit the extra spacing around parenthesis on one side of a function call, creating a lopsided appearance:

```
// watch out for this
foo (5 );
foo ( 5);

// should be
foo ( 5 );
```

The spacing rules in general prove difficult for some:

```
// this is a mistake
i = foo( ( a + b )/c );

// should be
i = foo (( a + b ) / c );
```

Try to learn the spacing rules and make them part of your muscle memory as you type.

Following these common mistakes are lapses in use of the divider comments and failure to respect alphabetization, particularly when methods are being added to an existing file.

Working Agreement – When to Reformat Code

Four simple rules for when to apply the coding styles outlined in this guide:

1. Always style new code to the best of your ability.
2. Do not re-style existing code unless you have a legitimate sprint task that requires you to rewrite or refactor it. We don't want you to spend your time sitting around re-styling code; if the code is working, leave it alone.
3. Whenever you see a code or header file that does not follow our file layout guidelines (i.e. sections dividers and alphabetization), update it to match the guidelines.
4. If you need to make a small change to a file (to add a method or class member, for example), match the existing coding style of the file to the best of your ability.

Formatting

Tabs vs. Spaces

Use tabs, not spaces. Set tabs to display as four (4) spaces in your IDE settings.

Visibility Qualifier Indentation

Always indent by a single tab below any visibility qualifier:

```
// Do not do this
   @protected
   NSInteger   mFoo;
   NSInteger   mBar;


// This is OK
public:
   u32   mFoo;
   u32   mBar;
```

Alphabetization

Always alphabetize sections of functions and methods, whether declarations or implementations. Do not group functions or methods by your opinion on how they might be associated or seem to you to 'go together.' The only system of order we desire for functions and methods is alphabetization.

Alphabetization of member variable declarations is optional. These may be grouped by your personal ideas about associations between them, or in any fashion.

If you are changing the code, always respect alphabetization. If you see sections of code where function declarations or implementations are not alphabetized, assume it is a mistake and fix their order. If you need to refactor code or rename methods, fix their alphabetical order immediately upon renaming.

Alignment

You will frequently see sections of our code grouped into a columnar alignment based on tabs. While alignment is encouraged, the only place where it is mandatory is in sections of function or method declarations (as described later in this document).

Of course, you may align any part of your code that makes sense to you or seems to improve readability. If you choose to align your code, use tabs instead of spaces.

```
// This is OK
   u32            mFoo;
   string         mBar;
   list < u32 >   mBaz;

// This is OK
   void           FuncA          ( u32 foo, u32 bar );
   void           FuncB          ( string foo );
```

Dividers

Code sections are delineated by dividers. Dividers should appear before anything having to do with the section or paragraph they belong to. Pay special attention to where these appear and make sure they appear in your code as well. Take care not to omit them or add sections without them.

The ‘thick’ divider looks like this:

```
//=============================================================//
// Title
//=============================================================//
```

The ‘thin’ divider looks like this:

```
//-------------------------------------------------------------//
```

Section specific comments always go below their divider, never inside or above:

```
//=============================================================//
// Title
//=============================================================//
// Comments about this section

//-------------------------------------------------------------//
// Comments about this section
```

C++ Class Declaration

Indent each section of an the class declaration by a single indentation level. Do not indent privacy scoping keywords. Alignment of member variable declarations is optional. Always align the names and parameters of any member method declarations. You may use different alignments for different sections of method declarations.

Always place a thick divider above the class declaration and a thin divider above any section of method declarations.

```
// This is OK

//=============================================================//
// Foo
//=============================================================//
class Foo {
private:

   int   mFoo;
   int   mBar;

public:

   //-------------------------------------------------------------//
   void   methodOne    ( int p1, int p2 );
   void   methodTwo    ( int p1, int p2 );
};
```

If the class contains static C functions for export to Lua, prefix these functions with an underscore, name them using camelCase and place them in their own, private declaration block with a thin divider above them:

```
// This is OK

//===================================================================//
// Foo
//===================================================================//
class Foo {
private:

   int   mFoo;
   int   mBar;

   //-------------------------------------------------------------------//
   static int  _luaFunctionOne   ( lua_State* L );
   static int  _luaFunctionTwo   ( lua_State* L );


public:

   //-------------------------------------------------------------------//
   void   methodOne    ( int p1, int p2 );
   void   methodTwo    ( int p1, int p2 );
};
```

Class sections should first be private or protected then public. The only exception to this rule is when a public section must absolutely be declared before e private or protected section to make the values of internal enumerations or constants available. In this case, use two public sections, one above and one below. Place only the minimum required members in the top section:

```
// This is OK

//===================================================================//
// Foo
//===================================================================//
class Foo {
public:

   enum {
      THING_ONE,
      THING_TWO,
      THING_THREE,
      TOTAL_THINGS,
   };

private:

   Thing   mThings [ TOTAL_THINGS ];
public:

   //-------------------------------------------------------------------//
   void   methodOne    ( int p1, int p2 );
   void   methodTwo    ( int p1, int p2 );
};
```

C++ Class Implementation

The main section of a class implementation should have a thick divider containing the class name as its header. Following this, each method implementation should be provided alphabetically, with a single thin divider immediately preceding it.

```
//===================================================================//
// Foo
//===================================================================//
```

```
//---------------------------------------------------------------//
void Foo:methodOne ( int p1, int p2 ) {
}

//---------------------------------------------------------------//
void Foo:methodTwo ( int p1, int p2 ) {
}
```

If the class has a Lua API, place this in its own section above the class implementation. If you require local functions, these two should be place in their own section. All functions in all sections must be alphabetized.

```
//===============================================================//
// local
//===============================================================//

//---------------------------------------------------------------//
void localFunction () {
}

//===============================================================//
// lua
//===============================================================//

//---------------------------------------------------------------//
int Foo:_luaFunctionOne ( lua_State* L ) {
   return 0;
}

//---------------------------------------------------------------//
int Foo:_luaFunctionTwo ( lua_State* L ) {
   return 0;
}

//===============================================================//
// Foo
//===============================================================//

//---------------------------------------------------------------//
void Foo:methodOne ( int p1, int p2 ) {
}

//---------------------------------------------------------------//
void Foo:methodTwo ( int p1, int p2 ) {
}
```

Obj-C Interface Indentation and Alignment

Indent each section of an interface declaration by a single indentation level. Do not indent the visibility qualifier. Alignment of member variable declarations is optional. Always align the names and parameters of any member method declarations. Always place a thick divider above the interface declaration and a thin divider above the method declarations.

```
// This is OK

//===============================================================//
// Foo
//===============================================================//
@interface Foo : NSObject {
@private
   NSInteger   mFoo;
   NSInteger   mBar;
}
```

```
   //---------------------------------------------------------------//
   -( void )   methodOne      :( int )p1 param2:( int )p2;
   -( void )   methodTwo      :( int )p1 param2:( int )p2;
@end
```

Preprocessor Indentation

Indent preprocessor conditionals as though they were any other part of the code.

```
void Foo () {

   #ifdef _DEBUG
      Log::Print ( “debug build\n” );
   #else
      Log::Print ( “release build\n” );
   #endif
}
```

Block Comments

Avoid block comments in code. Block comments are only to be used in file-level documentation blocks or in class declarations as required by a documentation tool such as Doxygen.

Line Length

We suggest your line lengths not exceed 80 characters, however you may choose to ignore this limit if you feel that doing so will improve readability. In general, optimize for vertical space over horizontal.

Extra Spaces

Include spaces after brackets and parentheses. Omit spaces between groups of brackets or parentheses:

```
int i = (( a + b ) * c ) - d [ 19 ];
int i = ( int )( floatFunc ());
int i = getFuncPtr ()( foo );
```

This applies to template parameters as well:

```
int i = foo < int >( bar );
```

The only exception to this rule is due to a shortcoming of C++’s grammar in which an extra space must be left between template parameter braces to avoid confusion with the bitwise shift operators:

```
// note the space between ‘> >’ to differentiate from ‘>>’
list < vector < int > > vecList;
```

Add a space after a function names and array names when calling or indexing:

```
// This is OK
Foo ( bar, baz );
int i = table [ j ];
int i = table [ j ][ k ];
```

```
// Do not do this
Foo( bar, baz )
int i = table [ j ] [ k ];
```

Obj-C Parameter Spacing

If you are writing or calling a function using that uses keywords for each parameter, do not put any spaces around the colons.

```
// This is OK
-( void ) initVecWithX:( int )x y:( int )y z:( int )z;
[ vec initVecWithX:1 y:2 z:3 ];
```

If you are writing or calling a function that omits additional keywords, then add a space before each parameter:

```
// This is OK
-( void ) initVec :( int )x :( int )y :( int )z;
[ vec initVec :1 :2 :3 ];
```

C++ Parameter Spacing

Always add an extra space after each comma.

```
// Do not do this
int i = Foo(x,y,z);

// Do not do this
int i = Foo( x,y,z );

// This is OK
int i = Foo ( x, y, z );
```

Multi-Line Obj-C Method Call

When breaking a long method call, place a single keyword and parameter pair on each line. Leave the object on the first line. Do not break the list unevenly or indent arguments more than one tab. Respect normal indentation and brace rules for the enclosing square brackets.

```
// Do not do this
int i = [ foo funcWithParam:param1
                  nextParam:param2 finalParam:param3 ];

// This is OK
int i = [ foo
   funcWithParam:param1
   nextParam:param2
   finalParam:param3
];

// This is OK
int i = [ foo
   funcWithParam:param1
   nextParam:param2
   [ bar funcWithParam:param1 otherParam:param2 ],
   finalParam:param4
];
```

If you have nested calls that you also wish to break up, indent them like any other scope.

```
// This is OK
```

```
int i = [ foo
   funcWithParam:param1
   nextParam:param2
   [ bar
      funcWithParam:param1
      otherParam:param2
   ]
   finalParam:param4
];
```

If you are calling a function that omits keywords, leave the method name on the first line of the call:

```
// Do not do this
int i = [ foo
   noKeywords:param1
   :param2
   :param3
];

// This is OK
int i = [ foo noKeywords
   :param1
   :param2
   :param3
];
```

You may place multiple parameters on one line if they naturally group, such as object-key pairs for NSDictionaries or printf-style format strings and matching arguments.

Multi-Line C++ Function Argument Lists

If you choose to break up an especially long argument list for a function call, do not break the list unevenly or indent arguments more than one tab. Respect normal indentation and brace rules for the argument list delimiters. If you have nested calls that you also wish to break up, indent them like any other scope.

```
// Do not do this
int i = MultiLineFuncExample ( param1, param2, param3,
                               param4, param5 );


// This is OK
int i = MultiLineFuncExample (
   param1,
   param2,
);

// This is OK
int i = MultiLineFuncExample (
   param1,
   param2,
   NestedFunc ( param1, param2 ),
   param4
);

// This is OK
MultiLineFuncExample (
   param1,
   param2,
   NestedFunc (
      param1,
      param2
   ),
   param4
);
```

Multi-Line Obj-C Method Declarations

After the method type, place each keyword and parameter pair on its own line, indenting a single level from the declaration. Place the semicolon immediately after the final pair

```
// This is OK
-( void )
   someMethod:( int )param1
   nextParam:( int )param2
   finalParam:( int )param3;
```

If you have chosen to omit keywords, place each parameter on its own line. Place the semicolon immediately after the final parameter

```
// This is OK
-( void ) noTokens
   :( int )param1
   :( int )param2
   :( int )param3;
```

If you have chosen to omit keywords and align parameter lists, leave the first parameter on the same line as the method name and each subsequent parameter on its own line. Place the semicolon immediately after the final parameter. You may add a line of whitespace above or below the declaration to improve readability.

```
// This is OK
-( void )      noTokensA      :( id )param1 :( int )param2;
-( void )      noTokensB      :( int )param1
                              :( int )param2
                              :( int )param3;
-( void )      noTokensC      :( string )param1;
```

Multi-Line C++ Function Declarations

Break the argument list over multiple lines, but use the indentation level of the aligned argument lists for the opening and closing parentheses, and indent the argument declarations one level further.

```
void        AlignedA    ( u32 param );
string      AlignedB    (  u32 param1,
                           u32 param2,
                           u32 param3
                        );
u32         AlignedC    ( string param );
```

Braces

Place the opening brace of any scoped statement on the same line of the statement. Indent the contents of the scope by a single level. Align the closing brace with the statement.

```
// This is OK
if ( flag ) {
   printf ( “hello\n” );
}

// This is OK
class Foo {
```

```
   u32 mBar;
};

// This is OK
void Func () {
   printf ( “func\n” );
}
```

Do not put statements of any kind on the same line as a closing brace.

```
// Do not do this
if ( flag ) {
} else {
}

// This is OK
if ( flag ) {
}
else {
}
```

Braces for if/else

Always use braces for if/else statements:

```
// Do not do this
if ( foo ) {
   if ( bar )
        printf ( “foo && bar\n” );
}
else
   ( “!( foo || bar )\n” );

// This is OK
if ( foo ) {
   if ( bar ) {
        printf ( “foo && bar\n” );
   }
}
else {
   ( “!( foo || bar )\n” );
}
```

You may omit the braces from an ‘if’ statements only if all three of the following conditions are true:

1. There is no ‘else’ clause.
2. The body of the ‘if’ is a single ‘break,’ ‘continue’ or ‘return statement.
3. The body of the ‘if’ is on the same line as the ‘if’ statement.

```
// Do not do this
if ( foo )
   return;

// Do not do this
if ( foo ) return;
else printf ( “bar\n” );

// This is OK
if ( foo ) return;
```

Whitespace for Operators

Ensure there is a single space around any binary operators, including logical, bitwise and assignment operators.

```
// Do not do this
foo=bar+baz;

// This is OK
foo = bar + baz;
```

Omit the space immediately to the right of any unary operator (before the operand):

```
// Do not do this
foo = ~ bar;
foo = - bar;
foo = ! flag;

// This is OK
foo = ~bar;
foo = -bar;
foo = !flag;
```

Casting is formatted as a unary operator:

```
int i = ( int )( foo / 12.0f );
int i = ( int )bar;
```

Pointer and Reference Declaration

When declaring a pointer or reference variable, align the designator with the type name.

```
// Do not do this
Foo *foo;
Foo &foo;

// This is OK
Foo* foo;
Foo& foo;
```

The spacing rules for unary pointer operators are the same as all unary operators:

```
// This is OK
foo = *bar;
foo = &bar;

// Do not do this
foo = * bar;
foo = & bar;
```

Obj-C File Layout

The overall order for the contents of your header files should be:

```
Copyright notice
Imports
Global constants
Global variable externs
Forward declarations
Obj-C Class declarations
        Members section
        Methods section
                Property directives (alphabetical)
```

```
            Methods (alphabetical)
```

The overall order for the contents of your source files should be:

```
Copyright notice
Imports
File global constants
Globals
File globals
Obj-C class definitions
        Nameless category declaration for private methods
        Class definition
                Synthesize directives (alphabetical)
                Methods (alphabetical)
                Protocol implementations (alphabetical)
                        Implemented protocol methods (alphabetical)
```

The sections marked as ‘alphabetical’ should be alphabetized within the file. As new methods are added or methods are deleted, reorder the methods to keep them alphabetized.

Naming Conventions

Function Parameters and Local Variables

Always use camelCase:

```
// This is OK
int localVar;
void foo ( int bar, int baz );
```

Variables of type lua_State* are an exception:

```
lua_State* L;
```

Member Variables

Always use TitleCase prefixed with a lowercase ‘m’:

```
// This is OK
int mMemberVar;
```

Local use of C++ Class Members

Access local class members using the ‘this’ pointer explicitely:

```
// local use of class member in method implementation
this->mSomeMember = this->SomeFunc ();
```

Local use of C++ Static Class Methods

When using a static method in a non-static member function of that same class, prefix the static method name anyway:

```
// local use of static member function
void SomeClass::SomeFunc () {

   int i = SomeClass::SomeStaticFunc ();
   this->mSomeMember = this->SomeMemberFunc ( i );
}
```

Globals

Always use TitleCase prefixed with a lowercase ‘g’:

```
// This is OK
int gGlobalVar;
```

Static Locals

Always use camelCase:

```
// This is OK
static int localStaticVar;
```

Static Globals

Always use TitleCase prefixed with a lowercase ‘s’:

```
// This is OK
int sStaticGlobal;
```

C++ Static Members

Always use TitleCase prefixed with a lowercase ‘s’:

```
// This is OK
int sStaticMember;
```

C++ Templates

Always use ALL_CAPS for C++ template parameters. Place the ‘template’ keyword and parameter list on its own line.

```
// This is OK
template < typename PARAM_TYPE, typename RETURN_TYPE >
RETURN_TYPE Foo ( const PARAM_TYPE& type );
```

Constants

Use or ALL_CAPS separated with underscores:

```
// This is OK
#define CONST_NAME 5
static const u32 CONST_NAME = 5;
```

Obj-C Enumerations

Choose a unique type name for the enumeration. Use TitleCase. Name the enumeration using the unique name. Prefix each member of the enumeration with the unique name.

```
// This is OK
enum EnumName {
   EnumNameAlpha,
   EnumNameBeta,
   EnumNameGamma,
};
```

If the enum is closely associated with a class, use the class name along with a modifier as the unique name.

```
// This is OK
enum TableStyle {
   TableStyleUgly,
   TableStyleUglier,
   TableStyleUgliest,
};
```

You may optionally apply the convention for constants to the enum members:

```
// This is OK
enum EnumName {
   ENUM_NAME_ALPHA,
   ENUM_NAME_BETA,
   ENUM_NAME_GAMMA,
};
```

Enumerations should always leave a trailing comma after the final member.

```
// Do not do this
```

```
enum EnumName {
   EnumNameAlpha,
   EnumNameBeta,
   EnumNameGamma
};

// This is OK
enum EnumName {
   EnumNameAlpha,
   EnumNameBeta,
   EnumNameGamma,
};
```

Global C++ Enumerations

There are several acceptable styles for global enums under C++.

You may follow the enum naming convention described for Objective-C.

You may package the enum in a containing class, struct or namespace. If you do so, omit the enum name from its members and instead use one of the naming conventions for constants:

```
// This is OK
namespace EnumName {
   enum {
      ALPHA,
      BETA,
      GAMMA,
   };
};
```

If the container exists only to hold the enum, use word ‘Type’ as the typename for the enum:

```
// This is OK
namespace EnumName {
   enum Type {
      kAlpha,
      kBeta,
      kGamma,
   };
};
```

If the container is a class or namespace that holds multiple enums, use descriptive names for the enum type or omit enum types altogether. In any event, use the naming convention for constants for the enum’s members.

```
// This is OK
class Foo {

   enum Level {
      ONE,
      TWO,
      THREE,

   };

   enum Difficulty {
      EASY,
      MEDIUM,
```

```
      HARD,
   };
};

// This is OK
class Foo {

   enum {
      kMaskOne    = 1 << 0,
      kMaskTwo    = 1 << 1,
      kMaskThree  = 1 << 2,
   };

   enum {
      kReady,
      kSet,
      kGo,
   };
};
```

Enumerations should always leave a trailing comma after the final enum member.

```
// Do not do this
enum EnumName {
   EnumNameAlpha,
   EnumNameBeta,
   EnumNameGamma
};

// This is OK
enum EnumName {
   EnumNameAlpha,
   EnumNameBeta,
   EnumNameGamma,
};
```

Type Names

Types (including classes, structs, interfaces, protocols, enumerations and typedefs) should always be named using TitleCase.

Class Name and Namespace Postfixes

The following class name postfixes usually have the following special meanings:

1. **Base:** An abstract base class or any class specifically intended to be inherited.
2. **Mgr:** A singleton object or a class containing only class methods meant to manage static state. For example ‘GfxDeviceMgr.’ May also be a namespace containing global variables.
3. **Shim:** A template class meant to bridge a class to a base class while provide a partial implementation for convenience. A shim will always accept a supertype as a template parameter.
4. **Util:** A class containing only class methods and no state or a namespace containing only functions. For example ‘StringUtil.’

Library Initials

Library initials are two to four capitalized letters meant to identify a library or framework. These letters do not have to be acronyms or contractions of a word: they are simply the ‘initials’ of the framework.

Library initials should be prefixed before any member of the global namespace provided that said member does not require a leading single letter prefix. For example:

```
MOAIClassName
MOAIEnumName
MOAIFuncName
```

In the case of global namespace members requiring a leading single letter prefix:

```
kMOAIConstValue
gMOAIGlobal
sMOAIStaticGlobal
```

The initials may be omitted from static globals:

```
sMOAIStaticGlobal // OK
sStaticGlobal // also OK
```

In the case of namespace members requiring ALL_CAPS, prefix the initials followed by an underscore:

```
MOAI_CONST_VALUE
```

Library initials need only be applied to multi-project source code. Project-specific source code should not use library initials:

```
// If in multi-project source code
MOAIName
kMOAIConstName
MOAI_CONST_NAME

// If in project-specific code
Name
kConstName
CONST_NAME
```

Acronyms

Acronyms should always follow the case style of the name that includes them:

```
UfoClassName // TitleCase
gUfoGlobalVar // camelCase
ufoLocalVar // camelCase
UFO_CONST_NAME // ALL_CAPS
```

Acronyms should not contradict the case style of the name that includes them:

```
UFOClassName // wrong
gufoGlobalVar // no
UfoLocalVar // do not
Ufo_CONST_NAME // bad
```

Obj-C Methods

Method names, keywords and parameters are always camelCase. You may use the 'Apple style' naming convention for methods with keywords.

```
// This is OK
-( void ) someFuncWithParam:( int )param1 paramTwo:( int )param2;
```

Alternatively, you may omit keywords and use the ‘C style’ naming convetion.

```
// This is OK
-( void ) someFunc :( int )param1 :( int )param2;
```

Obj-C Initializers

Always use ‘Apple style’ method naming for class initializers.

```
// Do not do this
-( void ) init :( int )param1 :( int )param2 :( int )param3;

// This is OK
-( void ) initWithParam:( int )param1 paramTwo:( int )param2;
```

Obj-C Protocols

As types, protocol names should be written in TitleCase and postfixed with the word ‘Protocol.’ Protocol methods should be prefixed by the protocol name in camelCase.

```
// This is OK
@protocol FooProtocol
   -( void ) fooProtocolMethodOne;
   -( void ) fooProtocolMethodTwo;
@end
```

If a protocol is used as a delegate for a particular class, prefix the protocol with the class name in TitleCase, and use the word ‘Delegate’ instead of ‘Protocol.’

```
// This is a delegate protocol for a class named ‘GSTable’
@protocol GSTableDelegate
   -( void ) gsTableDelegateMethodOne;
   -( void ) gsTableDelegateMethodTwo;
@end
```

C++ Methods and Functions

Always use TitleCase for the method name and camelCase for the parameter names.

```
// Do not do this
void func ( int ParamOne, int ParamTwo, inte ParamThree );

// This is OK
void Func ( int paramOne, int paramTwo, int paramThree );
```

C++ Accessor Methods

Use verb prefix. The following verb prefixes are reserved for C++ accessors:

1. **Get:** Returns a property.
2. **Is:** Boolean a Boolean property.
3. **Set:** Sets a property.

```
// This is OK
```

```
string   GetName        ();
bool     IsVisible      ();
void     SetName        ( string name );
void     SetVisible     ( bool visible );
```

Method Name Verbs

1. **Affirm:** Lazy initialization. If an object or member doesn't exist, it will be created and initialized. If it already exists, nothing is done. In the context of a collection, adds an object to the collection only if object is not already in the collection. Affirm may or may not have a return value.
2. **Alloc:** Creates a new instance of an object. The object will be retained (if applicable).
3. **Clear:** Releases resources associated with an object. If object is a container, removes all elements. Object should remain initialized and suitable for use after a call to a Clear method.
4. **Contains:** Reserved for collections. Boolean check to see if the set contains an object.
5. **Init:** Initialize an object. Obj-C: classes are not expected to allow re-initialization; C++: classes should handle re-initialization.
6. **Insert:** Add an object to a collection.
7. **New:** Returns a new and retained instance of an object.
8. **Release:** Reserved for reference counted objects. Decrements an object's reference count.
9. **Remove:** Remove an object from a collection.
10. **Retain:** Reserved for reference counted objects. Increments an object's reference count.