

IPSquad : The Way Of The Code

Java Coding Standard

Version 1.1

Author : Kévin Ottens

Table of Revisions

Revision	Date	Author(s)	Description
1.0	06/10/2002	Kévin Ottens	Document writing (mostly based on our C++ coding standard)
1.1	08/10/2002	Kévin Ottens	Little corrections (references to C++)

Table of Content

Preface	1
Part I : Make Names Fit	1
Chapter 1 : Selecting names	2
1. Class Names.....	2
2. Method and Function Names.....	2
3. Variable Names.....	2
3.1. Exceptions.....	3
4. No All Upper Case Abbreviations.....	3
4.1. Justification.....	3
Chapter 2 : Naming scheme	4
1. Class Names.....	4
2. Method Names.....	4
2.1. Justification.....	4
3. Class Member Names.....	5
3.1. Justification.....	5
4. Method Argument Names.....	5
4.1. Justification.....	5
5. Variable Names on the Stack.....	6
5.1. Justification.....	6
6. Static Variables.....	6
6.1. Justification.....	6
7. Static Constants.....	6
Part II : Code Formatting	8
Chapter 3 : Braces and parenthesis	9
1. Braces Policy.....	9
1.1. Justification.....	9
2. Braces Usage.....	9
2.1. Justification.....	9
3. Parenthesis Policy.....	10
Chapter 4 : Class Design	11
1. Required Class Methods.....	11
1.1. Details.....	11
1.2. Justification.....	11
2. Accessor Style.....	12
Chapter 5 : Class and file organization	13
1. File Header.....	13
2. Class Documentation.....	13
3. Class Layout.....	14

Part III : Language Directives and Best Practices.....	16
Chapter 6 : Uses Of The Constructor.....	17
1. Only initialize member variables.....	17
2. Delegate all logic.....	17
Chapter 7 : Best practices.....	18
1. Prefer positive boolean comparisons.....	18
2. Handle cleanup situations with boolean indicators.....	18
2.1. Justification.....	19

Preface

This document explains the Java coding style we've adopted and lists a number of best practices that should be followed when contributing to our Java projects.

Part I
Make Names Fit

Chapter 1

Selecting names

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

If you find all your names could be `Thing` and `DoIt` then you should probably revisit your design.

1. Class Names

- Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something `DownloadAgent` conveys real information.

2. Method and Function Names

Usually every method and function performs an action, so the name should make clear what it does : `checkForErrors()` instead of `errorCheck()`, `dumpDataToFile()` instead of `dataFile()`. This will also make functions and data objects more distinguishable. Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

Suffixes are sometimes useful :

- `Max` -to mean the maximum value something can have.
- `Cnt` -the current count of a running count variable.
- `Key` -key value.

For example : `retryMax` to mean the maximum number of retries, `retryCnt` to mean the current retry count.

Prefixes are sometimes useful :

- `is` -to ask a question about something. Whenever someone sees `Is` they will know it's a question.
- `get` -get a value.
- `set` -set a value.

For example : `isHitRetryLimit`.

3. Variable Names

Make every variable name descriptive, limit the use of abbreviations or letter-words. It's worth writing words completely since it makes the code much more readable. Beware however that when trying to find a good name, you don't end up with something like 'the_variable_for_the_loop', use a proper English word for it like 'counter' or 'iterator'. English is a rich language and trying to find a correctly fitting word is important for code brevity, cleanness and variation. Whenever in doubt, just use a thesaurus like Merriam-Webster (<http://www.m-w.com>) or a rhyming dictionary like Rhyme (<http://rhyme.sourceforge.net/>).

3.1. Exceptions

Some standard variables are used for often recurring tasks. Below is a list of those that are accepted :

- `i` : integer counter
- `it` : iterator
- `<type>_it` : iterator of a certain type for differentiation amongst types
- `tmp_<type>` : eg. `tmp_string`, `tmp_int`, `tmp_float` for variables that are solely used for the storage of temporary intermediate values

4. No All Upper Case Abbreviations

When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what.

4.1. Justification

People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable.

Take for example `NetworkABCKey`. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

Chapter 2

Naming scheme

A standard naming scheme is important to ensure that all code looks similar and that every developer can understand new code immediately without have to grasp a new naming scheme first.

One of the main aspects of this naming scheme is that all names should contain key information about the type of language construct is refers to. Additionally, certain prefixes will be used to prevent common error in the use of basic Java concepts such as scope. This however doesn't involve into a full-blown and difficult to understand and maintain Hungarian notation.

1. Class Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is upper case
- No underbars ('_')

Example 2.1: Class Names Example

```
class NameOneTwo  
class Name
```

2. Method Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is lower case
- No underbars ('_')

2.1. Justification

- Differentiates the first word part, which is often a verb. This makes it very clear what a method does.
- Not exactly similar to class names and thus makes `Class.doSomething()` much more readable as `Class.DoSomething()`, clearly indicating through case which is which.

Example 2.2: Method Names Example

```
public class NameOneTwo  
{  
    public int doIt()  
    {  
        ...  
    }  
}
```

```
        public void handleError()  
        {  
            ...  
        }  
    }  
}
```

3. Class Member Names

- Member names should be prepended with the character 'm'.
- Member the 'm' use the same rules as for class names.

3.1. Justification

- Prepending 'm' prevents any conflict with method names. Often your methods and attribute names will be similar, especially for accessors.

Example 2.3: Class Member Names Example

```
public class NameOneTwo  
{  
    private int mVarAbc;  
    private int mErrorNumber;  
    private String mName;  
  
    public int varAbc()  
    {  
        ...  
    }  
  
    public int errorNumber()  
    {  
        ...  
    }  
}
```

4. Method Argument Names

- The first character should be lower case.
- All word beginnings after the first letter should be upper case as with class names.

4.1. Justification

- You can always tell which variables are passed in variables.
- You can use names similar to class names without conflicting with class names.

Example 2.4: Method Argument Names Example

```
class NameOneTwo  
{
```

```
        public int startYourEngines(Engine someEngine, boolean autoRestart)
        {
            ...
        }
    }
```

5. Variable Names on the Stack

- Use all lower case letters
- Use '_' as the word separator.

5.1. Justification

- With this approach the scope of the variable is clear in the code.
- Now all variables look different and are identifiable in the code.

Example 2.5: Variable Names on the Stack Example

```
class NameOneTwo
{
    public int handleError(int errorNumber)
    {
        int error = osErr();
        Time time_of_error;
        ErrorProcessor error_processor;
        String tmpstring;
    }
}
```

6. Static Variables

- Static variables should be prepended with 's'.

6.1. Justification

- It's important to know the scope of a variable.

Example 2.6: Static Variables Example

```
class Test
{
    private static StatusInfo msStatus;
}
```

7. Static Constants

- Static constants should be all caps with '_' separators.

Example 2.7: Static Constants Example

```
class Test
{
    public static final int A_STATIC_CONSTANT = 5;
}
```

Part II

Code Formatting

Chapter 3

Braces and parenthesis

1. Braces Policy

Place braces under and inline with keywords, like this :

Example 3.1: Braces Policy Example

```
if(condition)           while(condition)
{                       {
...                   ...
}                       }
```

1.1. Justification

- If you use an editor (such as vi) that supports brace matching, this is a much better style than the default unix style where braces aren't vertically aligned. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace.

Example 3.2: Braces Policy Justification

```
if(very_long_condition && second_very_long_condition)
{
...
}
else if(...)
{
...
}
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

2. Braces Usage

All if, while and do statements must either have braces or be on a single line.

Always Uses Braces Form, even if there is only a single statement within the braces.

2.1. Justification

- Easier to read, you just have to scan for one form.
- Uniform idiom for scope blocks since they are all enclosed in braces.
- It provides a more consistent look.
- This doesn't affect execution speed and it's easy to apply.
- It ensures that when someone adds a line of code later there are already braces and they don't forget.

Example 3.3: Brace Usage Example

```
if(somevalue == 1)
{
    somevalue = 2;
}
```

3. Parenthesis Policy

- Do put parens next to keywords.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

Example 3.4: Parenthesis Policy Example

```
if(condition)
{
}

while(condition)
{
}

s1 = s.clone();

return 1;
```

Chapter 4

Class Design

1. Required Class Methods

To be good citizens almost all classes should implement the following methods. If you don't have to define and implement any of the "required" methods they should still be represented in your class definition as comments. If you just let the compiler generate them without indicating through comments that you know that this is the intended behaviour, people might wonder about the possibility of an omission or oversight.

1.1. Details

1.1.1. Default Constructor

If your class needs a constructor, make sure to provide one. You need one if during the operation of the class it creates something. This includes creating memory, opening file descriptors, opening transactions etc.

If the default constructor is sufficient add a comment indicating that the compiler-generated version will be used.

If your default constructor has one or more optional arguments, add a comment indicating that it still functions as the default constructor.

1.1.2. Copy constructor

If your class is copyable, define a copy constructor.

1.2. Justification

A default constructor allows an object to be used in an array.

A copy constructor ensure an object is always properly constructed.

Example 4.1: Required Class Methods Example

```
class Planet
{
    public Planet()
    {
        this(5);
    }

    public Planet(int radius)
```

```
    {  
        ...  
    }  
    public Planet(Planet from)  
    {  
        ...  
    }  
};
```

2. Accessor Style

Accessor methods provide access to the attributes of an object. Accessing an object's attributes directly, as is commonly done in C structures, is greatly discouraged in Java. It exposes implementation details of the object and degrades encapsulation.

The way to implement accessors accepted is the standard in Java API, it's the `get/set` couple of methods.

Example 4.2: get/set accessor style

```
class X  
{  
    private int mAge;  
    public int getAge()  
    {  
        return mAge;  
    }  
    public void setAge(int age)  
    {  
        mAge = age;  
    }  
}
```

Using this approach, it's possible to include some checks about the value provided to the `setAge()` method.

The huge drawback here is that objects aren't treated in their own right and that encapsulation somewhat fails. We have chosen this style only to be conform to the standard Java coding style.

Chapter 5

Class and file organization

1. File Header

A common file header for the whole project is important from a legal point of view and quickly find file version.

Following is the template that should be used to organize each file header for a project using the GPL.

Example 5.1: File header for a project using the GPL

```
/*
 * $Id$
 *
 *
 *           [Project Name]
 *
 * Copyright (C) 2001,2002 [Main developers names]
 * IPSquad <team@ipsquad.tuxfamily.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */
```

2. Class Documentation

For the creation of developer API documentation we're using Doxygen. Since Doxygen understand javadoc syntax too, it's better to document as if we used javadoc to interact easily with other Java projects. The only exception to this rule is the use of the *grouping members* tags (which do not exist in javadoc).

For more information on javadoc comment, please refer to javadoc documentation itself. Now we'll focus on the *grouping members* feature.


```
// JAVA API IMPORTS
//

// REUSED COMPONENTS IMPORTS

// PROJECT IMPORTS
//

package PP;

class XX
{
    // MEMBER VARIABLES
    //
    private int mName;

    /// @name Constructors
    //@{

    // CONSTRUCTORS

    /**
     * Default constructor.
     */
    public XX()
    {
        ...
    }

    //@}

    // OPERATIONS

    /// @name Accessors
    //@{

    // ACCESS

    //@}

    /// @name Inquiries
    //@{

    // INQUIRY

    //@}
};
```

Part III
Language Directives and Best Practices

Chapter 6

Uses Of The Constructor

1. Only initialize member variables

The constructor should only initialize the member variables. Also, explicitly initialize all member variables even if you're just calling their default constructor. It's better to be clear from the beginning than in doubt later.

2. Delegate all logic

No real action should be done in the constructor, delegate everything to a separate `initialize()` method. This will allow multiple constructors to use the shared code logic in the `initialize` method by passing the appropriate arguments.

Chapter 7

Best practices

1. Prefer positive boolean comparisons

It's much easier to think in a positive way about a situation than to be presented with the negative alternative and having to transform it in your mind by yourself to positive. People tend to have a 'logical' or 'the default behaviour' feeling about true, which makes it easy to think about. On the contrary, false is mostly regarded as the 'exception', 'the error situation' or the 'alternative way out'. Therefore we prefer constructs like this:

Example 7.1: Positive boolean comparison, the right way

```
setup();
if(something == true)
{
    dowork();
}
cleanup();
return;
```

above the following negative counterpart :

Example 7.2: Positive boolean comparison, the wrong way

```
setup();
if(something == false)
{
    cleanup();
    return;
}
dowork();
cleanup();
return;
```

2. Handle cleanup situations with boolean indicators

Often you're presented with the problem that your code logic contains a series of initializations that can all potentially fail. Typically you want to interrupt any further execution, cleanup and return an error message. Such situations have been known to be resolved through the use of exceptions, `gotos`, large `if-then-else` constructs and boolean indicators. From these options, it's the last one we prefer.

Below is an example of such a typical code cleanup situation :

Example 7.3: Cleanup with boolean indicators

```
void someMethod()
{
    boolean file_setup = false;
    boolean dir_setup = false;

    /* try to create a new file object and open it for reading */
    File file = new File("/path/to/file");
    if(file.open(File.ReadOnly) == true)
    {
        file_setup = true;
    }

    String dir_path("/path/to/default/dir");
    if(file_setup == true)
    {
        /* if the file was setup, read its contents and use it for */
        /* further processing */
        TextStream textstream(file);
        String dir_path = textstream.readLine();
        dir_path = textstream.readLine();
    }

    /* try to create a new dir object and open it for reading */
    Dir dir = new QDir(dir_path);
    /* some vars that are needed by the dir logic */
    if(dir.exists() == 0)
    {
        /* do stuff with the dir */
        dir_setup = true;
    }
    else
    {
        System.out.println(dir_path + " couldn't be processed");
    }

    /* cleanup the dir setup if needed*/
    if(dir_setup == true)
    {
        /* cleanup what was done in the dir logic part */
    }

    /* cleanup the file setup if needed*/
    if(file_setup == true)
    {
        file.close();
    }
}
```

2.1. Justification

- You prevent unnecessary consecutive indentations as is the case with large if-then-else constructs.
- It's very easy and clear to follow the logical flow, no jumps are executed as with exceptions.

- You can perform context-sensitive cleanups that combine the states of several boolean indicators.