A Guide to Coding Style

Justus Piater <Justus.Piater@ULg.ac.be>

2005-01-14

Table of Contents

Abstract

This Style Guide summarizes useful coding standards, conventions, and guidelines for writing correct, high-quality, and maintainable code in C, C++, or Java. They are based on industry standards, although often reduced to a simplified form to teach the concepts yet keep the work load reasonable. By following these principles, you will be better able to produce code that is correct, requires less debugging effort, and is easier for the graders to follow.

In the real world, code conventions are important to programmers for a number of reasons:

- 80 % of the lifetime cost of a piece of software goes to *maintenance*. Hardly any software is maintained for its whole life by the original author. Code conventions improve the readability of the software, allowing software engineers to understand new code more quickly and thoroughly.
- Coding standards allow *teams of programmers* to more easily understand each others code and help spot errors. Rather than coding from scratch, team members are more likely to sharing and re-use code when the style and commenting conventions among team members are the same.

Some of the recommendations given in this document are generally accepted conventions and should be considered mandatory, while others are merely suggestions that may differ between styles. The ultimate goal is readability, and good is what contributes to it. In any case, there is no readability without *consistency of style*.

Note

This document uses object-oriented terminology and refers to “functions” as “methods”. For the purpose of this document, a “method” is the same as a “function”.

While the code examples are given in Java, the cited principles apply to all related languages.

1. Naming Conventions

The following general guidelines apply to *choosing identifier names* in your programs:

- Use full English descriptions for names. *Avoid using abbreviations*. For example, use names like `firstName`, `lastName`, and `middleInitial` rather than the shorter versions `fName`, `lName`, and `mi`.
- *Avoid unnecessarily long names*. For example, `setTheLengthField` should be shortened to `setLength`. If you feel that you cannot clearly express the meaning of a variable within a reasonable variable name (up to about 20 characters), you should think about restructuring your code.
- Avoid names that are very similar. For example, if you store a list of products in a variable called `perishableProducts`, it may be better to refer to a particular product as `thisPerishableProduct` (or `thisProduct` or `product`) rather than `perishableProduct` to avoid confusion.

Why use *English* names? The programming language keywords are in English, and it is difficult to read multiple languages in parallel. Even more importantly, in the real world barely any software code stays within the realm of a single language, forcing us to use English as the de-facto common denominator.

You should write your code with the aim of making it understandable to others. Others will need to read and understand your code and one of the major keys to understanding is through the use of meaningful identifier names.

By using meaningful names, you go a long way towards writing *self-documenting code*. That is, code that is understandable on its own without requiring accompanying comments. For example, look at the following code segment and determine for yourself that the variable names `salesTax` and `incomeTax` are preferable to `tax1` and `tax2` because the names are self documenting.

```
double tax1; // sales tax rate (example of poor variable name)
double tax2; // income tax rate (example of poor variable name)

double salesTaxRate; // no comments required due to
```

```
double incomeTaxRate; // self-documenting variable names
```

1.1. Variable Naming Conventions

Choose meaningful names that describe what the variable is being used for. *Avoid generic names* like `number` or `temp` whenever their purpose is not absolutely clear.

Compose variable names using *mixed case letters* starting with a lower case letter. For example, use `salesOrder` rather than `SalesOrder` or `sales_order`.

Use plural names for arrays. For example, use `testScores` instead of `testScore`.

Exception: `for` loop counter variables are often named simply `i`, `j`, or `k`, and declared local to the `for` loop whenever possible.

```
for (int i = 0; i < MAX_TEMPERATURE; i++) {
  boilingPoint = boilingPoint + 1;
}
```

These conventions are common practice in the Java development community and also the naming convention for variables used by Sun for the Java core packages. Writing code that follows these conventions makes variables easy for others to distinguish from other types. They are also increasingly used in C++ and C code.

1.1.1. Declaring and commenting local variables

- Do not declare several variables within the same statement (separated by commas), except for trivial cases.
- If the meaning and use of the variable is not clear, add an endline comment (`//`, or `/* ... */` in C) stating what the variable is used for and why. However, a better solution is to choose a meaningful name to avoid the need for the endline comment.
- Declare variables immediately before they are used, rather than declaring all variables at the top of the method.

  In pre-ISO-1999 C, declare all variables at the beginning of the innermost block that delimits its intended scope. It can sometimes be a good idea to open a new block specifically for declaring variables close and limited to their local use.
- Whenever possible, initialize the variable with its starting value in the declaration statement.

  In pre-ISO-1999 C, if the first use is far away from the variable declaration, it is perhaps an indication of poor code design.

1.2. Constant Naming Conventions

- Use `ALL_UPPER_CASE` in your constant names, separating words with the underscore character. For example, use `TAX_RATE` rather than `taxRate` or `TAXRATE`.
- *Avoid using literal numbers* in the code. Literal numbers like `27` that appear in the code require the reader to figure out what `27` is being used for. Consider using named constants for any number other than 0 and 1.


```
day = (3 + numberOfDays) % 7; // NO! uses literal numbers

static final int WEDNESDAY = 3;
static final int DAYS_IN_WEEK = 7;

day = (WEDNESDAY + numberOfDays) % DAYS_IN_WEEK; // Yes, self-documenting
```

1.3. Method Naming Conventions

Try to come up with *meaningful method names* that succinctly describe the purpose of the method, making your code self-documenting and reducing the need for additional comments.

Compose method names using *mixed-case letters*, beginning with a lower case letter and starting each subsequent word with an upper case letter.

Begin method names with a strong action verb (for example, `deposit`). If the verb is not descriptive enough by itself, include a noun (for example, `addInterest`). Add adjectives if necessary to clarify the noun (for example, `convertToAustralianDollars`).

Use the prefixes `get` and `set` for *getter* and *setter* methods. Getter functions merely return the value of a variable; setter functions change the value of a variable. For example, use the method names `getBalance` and `setBalance` to access or change the variable `balance`.

If the method *returns a boolean value*, use `is` or `has` as the prefix for the method name. For example, use `isOverdrawn` or `hasCreditLeft` for methods that return `true` or `false` values. Avoid the use of the word `not` in the boolean method name, use the `!` operator instead. For example, use `!isOverdrawn()` instead of `isNotOverdrawn()`.

These conventions are *common practice*. Writing code that follows these conventions makes methods easy for others to identify. Even though variables and methods both begin with a lower-case letter followed by mixed case, they can easily be differentiated from each other because method names begin with a verb and always are immediately followed with a set of parenthesis, for example: `moveAccount()`.

1.4. Type Naming Conventions

Class names are always written in mixed case like method names, but begin with an *upper-case* letter.

By analogy, `typedefs` of structured types should be named in the same way.

In C, `structs` and `enums` are typically named in all-lowercase, ending with `_t`. However, you will usually want to `typedef` such types.

A case could be made for naming `typedefs` of non-structured types according to C-style conventions, to distinguish them from structured types. However, I feel it increases readability greatly if *all* `typedef` names begin with an upper-case letter, to distinguish them from variable names:

```
class SomeClass { ... };

typedef struct node_t {
  void *content;
  struct node_t *next;
} Node;

typedef enum season_t { SPRING, SUMMER, FALL, WINTER } Season;
```

2. Commenting Conventions

Comments provide readers with the information helpful for understanding your program.

Use comments to provide overviews or summaries of chunks of code and to provide additional information that is not readily available in the code itself.

Comment the details of nontrivial or nonobvious design decisions; avoid comments that merely duplicate information that is present in and clear from reading the code. We will use *three types of comments*:

- File header comments providing identification information about the program and author.
- Single-line comments providing overviews or summaries of chunks of code.
- Trailing comments that provide information for one line of code.

In addition, the C style comments `/* ... */` may be used temporarily for commenting out blocks of code during debugging. Programs submitted for a grade should *not* have blocks of code commented out however. Either fix the code or remove it from the program.

For readability, always leave space around the comment delimiters, and format multi-line comments nicely.

Always comment your code in *English*. See the Naming Conventions for why.

2.1. File Header Comments

File header comments provide information pertaining to the file as a whole. This includes at least the date of creation, the name of the creator, and a brief description of the code contained therein.

In real life, file header comments also contain revision and copyright information.

2.2. Single-Line Comments

Use single-line comments to provide *brief summary comments* for chunks of code.

Precede single-line comments with a blank line and align the comment with the code it summarizes. Do not feel the need to comment every single line of code, rather summarize chunks of code between 3 to 7 lines in length.

In Java and C++, begin single-line comments with a *double slash* (`//`) that tells the compiler to ignore the rest of the line.

2.2.1. Rationale

By separating chunks of code by blank lines and preceding each chuck with an explanatory comment, readers (and yourself) can *quickly identify and understand sections* of code where specific actions are being performed.

Why limit such comments to one line? Well, there is no absolute limit. However, if you think you need more than one or two lines to document a piece of code, you should ask yourself if that comment should not rather go into the method or function header. Lengthy explanations are typically relevant to the *user* of a function (e.g., details of a search algorithm, numerical methods used), and as such belong into the function documentation.

Always *add comments while you are coding* rather than waiting until the program is finished. The summaries and reasons will be fresh and accurate rather than place-fillers that offer no real value.

```
// Compute the exam average score for the midterm exam
int sumOfScores = 0;
for (int i = 0; i < scores.length; i++)
  sumOfScores = sumOfScores + scores[i];
average = float(sumOfScores) / scores.length;
```

2.3. Trailing Comments

Trailing comments are used to provide an explanation for a *single line of code*. Begin trailing comments with a double slash (`//`) and place them to the right of the line of code they reference.

Trailing comments are used to explain tricky code, specify what abbreviated variable names refer to, or otherwise clarify unclear lines of code.

In general, see whether trailing comments are *really necessary*, or whether you can rewrite tricky or unclear code, use meaningful variable names, or otherwise enhance the self-documenting character of the code.

2.3.1. Rationale

Trailing comments are a throwback to the days of assembly language programming where single lines of code were impossible to understand without the use of comments.

Modern programming languages like Java allow programmers to write self-documenting code using meaningful variable names, eliminating the need for trailing comments in many cases.

Compare the following two pieces of code:

```
ss = s1 + s2 + s3; // add the three scores into the sum
a = float(ss) / 3; // calculate the mean of the scores

sumOfScores = score1 + score2 + score3;
meanScore = float(sumOfScores) / NUM_SCORES;
```

The second version is more readable even though there are no comments.

2.4. Documentation of Classes and Methods

Comments are not sufficent to provide a complete documentation of your program. In addition, every class and method need to have a description of its behaviour and usage. For Java, the *Sun JavaDoc [http://java.sun.com/j2se/javadoc/writingdoccomments/]* standard is a powerful technique to write comments and allows the programmer to automaticaly generate the *html* documentation. For C++ and C, Doxygen [http://www.doxygen.org/] is a powerful and elegant system for source documentation using the same philosophy and syntax.

Use JavaDoc or Doxygen to document your code.

Here is an example of a method documentation:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 *
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
  try {
    return getImage(new URL(url, name));
  } catch (MalformedURLException e) {
    return null;
  }
}
```

3. Formatting

Formatting refers to the indentation, alignment, and use of white space to lay out your program to increase its *readability* by others.

Consistency is the key to producing readable code. While many can argue to merits of 2 versus 3 spaces of indentation, placement of curly braces, etc., the real key is to adopt a *formatting style* and keep to it.

Note

In the real world, organizations adopting a standard for formatting *increase programmer productivity* by reducing variations and trivial decisions among teams of programmers. A uniform code format allows experienced programmers to *understand code at a quick glance*; they do not need to visually parse the code word by word and operator by operator to see what it does.

Note

A typical piece of code is *written once* but *read many times*.

3.1. Indentation and Braces

Use *two spaces* for indentation to indicate nesting of control structures.

Avoid the use of tabs for indentation. Tabs depend on your system's settings. What works for you may not work for others. Use spaces instead.

Use a *syntax-aware editor* such as emacs [http://www.gnu.org/software/emacs/] to do the indentation for you automatically.

Be consistent in the placement of opening and closing braces. Their position should correspond to the block levels they delimit.

```
public class HelloWorld {
  public void greetUser(int currentHour) {
    System.out.print("Good ");
    if (currentHour < AFTERNOON) {
      System.out.println("Morning");
    }
    else if (currentHour < EVENING) {
      System.out.println("Afternoon");
    }
    else {
      System.out.println("Evening");
    }
  }
}
```

If a block consists of a single instruction, it is permissible to omit the braces:

```
System.out.print("Good ");
if (currentHour < AFTERNOON)
  System.out.println("Morning");
else if (currentHour < EVENING)
  System.out.println("Afternoon");
else
  System.out.println("Evening");
```

```
for (int w = 0; w < with; w++)
  for (int h = 0; h < height; h++)
    for (int d = 0; d < depth; d++)
      a[w][h][d] = 0;
```

However, if a subordinate block contains more than a single line at the same level, use braces for better readability, even if they are not required:

```
for (int w = 0; w < with; w++) {
  for (int h = 0; h < height; h++) {
    for (int d = 0; d < depth; d++) {
      a[w][h][d] = 0;
```



```
      b[w][h][d] = 1;
      c[w][h][d] = 2;
    }
  }                // unnecessary, but more readable
}                  // unnecessary, but more readable

if (terminate) {
  // stop here
  break;
}                  // unnecessary, but more readable
```

3.2. White Space

Use blank lines and blank spaces to improve the *readability* of your code.

Use *blank lines to separate chunks of program code*. Chunks are logical groups of program statements (generally 3 to 7 lines in length) and usually preceded with a single-line summary comment. Use one blank line before every program chunk. Use two blank lines before the start of each new method.

Use *one blank space*

- on both sides of binary arithmetic and relational symbols: `c = a + b;`
- after commas
- after semicolons in `for` statements
- after keywords `for`, `if`, `do`, `while`, `switch`. (Exception: operators that are used like a function, such as `this`, `super` or `sizeof`.)

Do *not* insert extra spaces

- around selection operators: `member.data`, `node->next`, `vec[i]`
- around parentheses
- next to a unary operator: `*p++ = !a;`
- before punctuation

Note

While the latter is contrary to French typesetting conventions, keep in mind that we *program in English*.

3.3. Line Length

Avoid lines longer than 80 characters. When an expression will not fit on a single line of 80 characters, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Align the new line with the beginning of the expression at the same level on the previous line.

Note

Needless to say that you *NEVER* put more than one statement onto the same line.

4. Structuring the Code

4.1. Method Complexity

Keep methods short. As a general rule, if a method does not roughly fit into an editor window, you should probably split it up into smaller methods.

The primary exception to this rule is a large `switch` with a long list of `case` statements inside.

Keep nesting levels shallow. Blocks nested more deeply than about 3 levels usually make the code hard to understand. Again, you can solve this by moving inner blocks into separate methods.

4.2. Modularity

Restrict access to information to a minimum:

- Restrict the scope and visibility of each variable as much as possible. In C and C++, use `const` wherever applicable.
- Avoid global variables and functions. Where you cannot avoid them (e.g., in C), restrict their scope to the compilation unit using `static` wherever possible.
- Organize your code into reasonably small compilation units.
- In object-oriented programs, carefully choose the right protection level for each class and each of its members.
- In C and C++, use the `const` keyword wherever possible.

4.3. Clarity and Robustness of Code

Stick to *standards* to ensure maximal *portability* of your code.

- If possible, configure your compiler to only accept standard-compliant code. (For the Gnu Compiler Collection [http://gcc.gnu.org/], use the compiler command-line options `--std=c99 --pedantic`. Be careful though – this does not necessarily reject extensions conforming to *newer* standards; see the next point.)
- Before using any particular construct or library function, be sure that its use is covered by your chosen standard(s).
- Rely only on documented, standard behavior. *Never try out* how a method or programming construct behaves in borderline cases. The behavior you observe may be different on another system, or tomorrow on your system.[1]
- If you find that you must use a non-standard method (e.g., provided by your specific programming environment or software library), then separate standard from non-standard code as cleanly as possible to facilitate a replacement of the non-standard parts at a later time.

[1] This is an extremely important point. Failure to observe it has led, among others, to the current abundance of faulty Web pages containing non-standard HTML and JavaScript code that rely on specific bugs in Microsoft's Internet Explorer, instead of pressuring Microsoft to fix their code. Always fix problems at the root!

Avoid questionable or irritating constructs. Do not write

```
if (a = b)
```

even if it is correct in your situation.

In `switch` statements, account for all possible cases (e.g. by including a `default`), and signal an error if an impossible situation occurs. In general, try to *catch seemingly impossible situations* so you can diagnose problems later. Do not assume that such situations will not occur.

In general, *always* compile your code with all reasonably avoidable warnings turned on (compiler command-line options `-Wall -W` for the Gnu Compiler Collection [http://gcc.gnu.org/]), and make sure your code compiles without any warnings at all.

If your programming language allows it, *declare variables close to their first use*. Also, stick to the general rule

> *One variable for one purpose*.

Do not reuse variables for other purposes, unless their purpose is trivial (such as loop counters named `i`). It makes code much easier to understand. Moreover, the compiler's optimizer does a much better job at eliminating redundancies than you do.

This document evolving from a Java Style Guide [http://www.cwu.edu/%7Egellenbe/javastyle/] by Ed Gellenbeck, Central Washington University, and benefitted significantly from comments by Sébastien Jodogne.