# **Coding Convention**

Different modules specified in the design document are coded in the Coding phase according to the module specification. The main goal of the coding phase is to code from the design document prepared after the design phase through a high-level language and then to unit test this code.

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Some of conventions are :

**1. Order of procedures**

When you add a new procedure to a file, don't just type it wherever your cursor happens to be. Instead, place related procedures together. It is often helpful to put a block comment (e.g., starting with a row of 75 asterisks, or whatever style you use, so long as you are consistent) at the beginning of each group of related procedures. Such block comments divide the file into sections that are readily apparent to readers.

In general, put public methods before private ones in your files. Organize your file with helper methods (whether public or private) after the main entry points. This permits readers to read your code top-down, which is more comprehensible: the purpose of each piece of code, and how it fits into the whole, is obvious. A reader can forward-reference to just the specification, not the whole implementation, of a helper method. This doesn't mean you necessarily have to *write* your in a code top-down order, but do organize it that way for readers.

**2. Comments**

Every code file that you write (Java classes, Perl scripts, etc.) needs to have a comment at the top explaining exactly what it does and, if applicable, how to run it. Otherwise it will be a mystery to others — and perhaps to you when you return to it. In some cases one or two sentences will do; in many other cases, the description needs to be more complete. Every non-trivial procedure should also contain a brief comment saying what it does. In the case of Java, there should be valid Javadoc comments for every method (both public and private). Each parameter should be described as well unless they are completely obvious. Don't add useless comments that just repeat the name of the parameter and its type, or in which the @returns clause is essentially identical to the procedure summary. Comments should enlighten, not merely repeat.

When a comment is a sentence, start it with a capital letter, end it with a period, and use correct grammar. Strive to keep comments and code to 80 characters whenever possible. (Don't be slavish — an exception here and there is OK &mdash but lots of violations lead to less-readable code. Don't assume that everyone uses the same width screen as you do — I assure you they do not — but 80 columns is a generally-accepted industry standard.) This makes it possible to print the code in a readable fashion and also to read the code in a standard-sized window.

Do not comment out large blocks of code with /\* ... \*/; instead, prefix each line by // . Among other things, this does not lose the indentation of the original code; without that indentation, the commented-out code is much too hard to read and understand. It also makes it clear what is commented out and what is not, even when the code is printed or is viewed without color highlighting.

**3. Code copying**

In general, you should not copy code. It is easy to make a mistake when copying, even easier to forget to update some of the copies when editing other copies, and difficult for readers to understand the distinction (or lack thereof) among the versions. Rather than copying, it is often better to use hooks or to generalize the original version.

If you are forced to copy code, then it is *essential* that you indicate where you got the original version from; this is important for understanding the code and for giving credit where credit is due, and to not do so is intellectually dishonest. Furthermore, you should indicate the reason for the copying and how this version differs from the original, and clearly indicate every change that you have made, perhaps with a distinctive comment that is indicated in the prefatory comment, or perhaps by giving a command that can be run to get a diff of your version of code against the original. If the original code is still being maintained, you should periodically update your code with respect to the upstream version, and should document how to do this.

**4. Local variables**

Local variables should have the most restrictive scope possible. For instance, don't do this:

int x;

...

for () {

...

x = ...;

... }

Instead, do this:

for () {

...

int x = ...;

...

}

A *loop-carried dependence* is when a variable is (sometimes) set on one loop iteration and used on the next iteration. A loop-carried dependence is the only reason to declare, external to a loop, a variable that is set in the loop. Reducing scopes makes it clear that there are no loop-carried dependences. Likewise, if two loops both use a temporary variable, you should declare two separate variables rather than reusing the same one, to indicate that there are no inter-loop dependences.

**5. Initialization**

Every variable and field should be explicitly initialized (set to an initial value). However, it should never be redundantly initialized to a temporary value that will not be read.

**Initialization for variables**

If a variable is initialized after its declaration but before it is used, it should not be initialized to a temporary value that will never be read. An example is

int x; // it would be bad style to initialize x to a dummy value

if (p) {

x = someValue;

} else {

x = otherValue;

}

It is clearer, when possible, not to reassign values immediately. Prefer the above construct with an else clause over

int x = otherValue; // this is confusing; put it in an else clause instead

if (p) {

x = someValue;

}

**Initialization for fields**

Fields should also be initialized exactly once. If a field is initialized by the constructor, then its declaration should *not* initialize it to a temporary value that will never be read.

* If a declaration has an initializer, that should be its initial value.
* If a declaration has no initializer, that is a signal to the programmer to look elsewhere for the initial value (and that the initial value differs for different instantiations).

In some languages (for example, Java), it is possible to omit the initializer for a field: boolean myField; is equivalent to boolean myField = false;. The short version is no more efficient, but it is more confusing. A reader must waste time searching the code (including in subclasses) to determine the initial value. The code is clearer if the initializer is explicit. Do so for all datatypes, including objects whose default value (in the absence of an initializer) is null.

# **Java Naming conventions :**

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

### Class :

* It should start with the uppercase letter.
* It should be a noun such as Color, Button, System, Thread, etc.
* Use appropriate words, instead of acronyms.
* **Example: -**

**public** **class** Employee

{

//code snippet

}

### Interface :

* It should start with the uppercase letter.
* It should be an adjective such as Runnable, Remote, ActionListener.
* Use appropriate words, instead of acronyms.

**Example: -interface** Printable{

  //code snippet

}

### Method :

* It should start with lowercase letter.
* It should be a verb such as main(), print(), println().
* If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
* **Example:-**

**class** Employee

{

//method

**void** draw()

{

//code snippet

}

}

### Variable :

* It should start with a lowercase letter such as id, name.
* It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore).
* If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
* Avoid using one-character variables such as x, y, z.
* **Example :-**

**class** Employee

{

//variable

**int** id;

//code snippet

}

### Package :

* It should be a lowercase letter such as java, lang.
* If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

**Example :-**

**package** com.javatpoint; //package

**class** Employee

{

//code snippet

}

### Constant :

* It should be in uppercase letters such as RED, YELLOW.
* If the name contains multiple words, it should be separated by an underscore(\_) such as MAX\_PRIORITY.
* It may contain digits but not as the first letter.
* **Example :-**

**class** Employee

{

//constant

**static** **final** **int** MIN\_AGE = 18;

//code snippet

}

## CamelCase in java naming conventions :

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.