Java Code Conventions

(Created by Sun Microsystems in 1997, changed by Frank Mehler 2022)

1 Why Have Code Conventions

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 engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

2 Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class.

2.1 Beginning Comments

All source files should begin with comment that lists the programmer(s), the date and version, and also a brief description of the purpose of the program. In addition the open points, i.e. still open problems of the class should be given.

```
/**
* Firstname Lastname
* Date, Version info
* The Example class provides ...
* Open points/ ToDo: ...
*/
```

2.2 Comments for Functions and Methods

All functions and methods should begin with a comment that gives the description of the function/method and a brief description of all input/output parameters with the @param tag and @return tag. These tags can be generated automatically in Eclipse: Position the mouse cursor in the function/ method and press SHIFT-ALT-J

```
     /**
      * Diese Methode berechnet die Kennzahl zu xyz
      * @param N Hoechster Wert der Folge
      * @return Ergebnis der Berechnung zu xyz
      */
     public static long ermittleKennzahl (int N) {…
```

Note: Getter and Setter-methods do not need comments

2.3 Class and Interface Declarations

This paragraph is not for Java beginners and can only be used after the introduction to Java classes, attributes and methods. The following table describes the parts of a class or interface declaration, in the order that they should appear.

1. Beginning Comments (`/**...*/`).
2. `package` statement. After that, `import` statements if necessary
3. `class` or `interface` statement
4. (Optional: Implementation comment (`/*...*/`) with details of the implementation)
5. Class (`static`) variables First the `public` class variables, then the `protected`, and then the `private`.
6. Instance variables First `public`, then `protected`, and then `private`.
7. Constructors
8. Methods: These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.
9. The `main`-method should be the last method at the bottom of a class.

3 Indentation (= Einrückung), Line Length, Wrapping Lines

Please use the Formatter Tool in Eclipse! Right mouseclick, Source, Format. Ctrl Shift F

Especially lengthy lines (> 120 chars) are not good for reading on smaller screens.

4 Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment. Not every single line of code needs to be commented!

Programs can have the following styles of implementation comments: block, single-line and end-of-line.

4.1 Block Comments

Block comments are used to provide descriptions of methods, data structures and algorithms. Block comments should be used before each method and describe the purpose of a method and the input/output parameters. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk "*" at the beginning of each line except the first.

```
/*
 * Here is a block comment.
 * This is the second line.
 */
```

4.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. Here's an example of a single-line comment in Java code:

```
if (a == 5) {
      /* Handle the condition. */
}
```

Alternatively, also // comments are possible as single-line comments, e.g.

```
if (a == 5) {
      // Handle the condition.
}
```

4.3 End-Of-Line Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting. Here's an example:

```
if (a == 2) {
      return true;          // special case
} else {
      return isPrime(a);    // works only for odd a
}
```

5 Declarations

5.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo, fooarray[]; //WRONG!
```

5.2 Placement

Variables should be as local as possible, i.e. try not to declare all variables at the beginning of a class, but normally within methods or blocks.
Put declarations for variables only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".)

```
void computeTemperature() {
      int int1; // beginning of method block
      if (condition) {
            int int2; // beginning of "if" block
      }
}
```

Example: The variables of `for` loops, which in Java can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++)
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
func() {
      if (condition) {
            int count; // AVOID!
      }
}
```

5.3 Initialization and Conversion

Try to initialize local variables (= only used locally within methods) where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Further initialization rules:

- Object attributes (= instance attributes) should be initialized in the Constructor, not after declaration
- Static attributes should be initialized directly after the declaration

Initialization rules for constants (= final)

- final object attributes in the Constructor, not after declaration

- final static attributes should be initialized directly after the declaration

Conversion between different datatypes is preferred via conversion functions, e.g.:

```
int myInt = Integer.parseInt(myString);

String myString = String.valueOf(myInt);
```

Narrowing typecasting (e.g. from `double` to `int`) should be avoided due to loss of information; if still used, then explanations (comments) are required.

5.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement.

```
class Sample extends Object {
   private int var1;
   private int var2;

   public Sample(int i) {
         this.var1= i;
   }
}
```

6 Statements

6.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++; argc--; // AVOID!
```

6.2 if and if-else Statements

```
if (condition) {

}

if (condition) {

} else {

}
```

For the sake of readability, please avoid nested statements with more than two if-Statements.

6.3 for Statements

A `for` statement should have the following form:

```
for (int i = 0; i < sortedData.length; i++) {

}
```

6.4 break

Java offers different possibilities to terminate a loop; the preferred method is the termination via the condition at the beginning of a loop, e.g.:

```
while (i  < 10 && !finished)

for (int i = 0; i < sortedData.length && !found; i++).
```

The use of a `break`-statement should be avoided, because the code often gets less readable and predictable. If there is a need for a break, a comment should describe why the normal termination via condition is not applicable or cumbersome.

6.5 While Statements, do-while Statements, switch Statements, try-catch Statements

All those statements should be formatted according to Eclipse formatter (see Templates). In addition, please avoid too many nested statements, i.e. normally three ore more nested loops should be separated in several methods. Please always use default-cases in switch Statements.

7 White Space

7.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related. One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

7.2 Blank Spaces

Blank spaces should be used in the following circumstances: A keyword followed by a parenthesis should be separated by a space. Example:

```
while (condition) {

}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

A blank space should appear after commas in argument lists.

Note: Blank spaces are automatically added correctly by Eclipse formatter

8 Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it's a constant, package, or class—which can be helpful in understanding the code.

Identifier Type	**Rules for Naming**	Examples

Variables	Variables are in mixed case (= camel case) with a lowercase first letter
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.
	Using variable names like `myRational` or `myDate` are only used without semantic context, e.g. in general examples of slides. If there is a context, the name should reflect the intention, e.g. `startDate` or `endDate`
		`int i;`
		`char c;`
		`float widthOfRectangle;`
		`Date birthDate;`

Constants	Constants should be all uppercase with words separated by underscores ("_").
	In contrast to the original Java-guidelines also local constants within methods should follow this convention.
		`static final int MIN_WIDTH = 4;`
		
		`final int MAX = 10;`

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized.
	Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
		`class Raster;`
		`class ImageSprite;`

Methods (and functions)	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();`
		`runFast();`
		`getBackground();`

Parameters in methods and functions	Parameters in methods/functions should be in mixed case (= camel case) with a lowercase first letter, even if the method-call uses a constant value.
	Only in case of constant parameters, uppercase is used.
		`berechneMittelwert (int anzahlElemente)`
		
		`zeileAusgeben (final String KOPFZEILE)`

9 Programming Practices

9.1 Providing Access to Instance and Class Variables

This paragraph is not for Java beginners and can only be used after the introduction to object oriented structures, i.e. Java classes consisting of attributes and their methods.

The recommendation is: Don't make any instance or class variable public without good reason. You only provide Setter- or Getter-methods if these methods are really needed and could be used.

9.2 Constants

Numerical constants (literals, e.g. 999 or -15) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

9.3 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
i = j = 3; // AVOID!
```

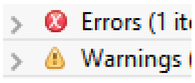
If primitive variables are used as input-parameters of methods or functions, please keep in mind, that Java uses "Call by value". Changes within methods/functions of primitive input parameters are avoided. Instead, return-parameters are used for delivering new or changed values.

9.4 main-Method

The `main`-method should be placed at the end of a class, not the beginning. A main-method should not contain business-logic or lengthy calculations (that's what methods and functions are for). A `main`-method should be short, i.e. only a few lines.

9.5 Eclipse errors and warnings

Eclipse gives errors and warnings. Obviously, errors must be corrected immediately. Warnings provide hints for improvements and should be corrected at least in the final version.



10 Java Source File Example

The following example shows how to format a Java source file containing a single public class.

```java
/**
* Firstname Lastname
* Date, Version info
* The Blah class provides ...
* Open points/ ToDo: implementation of methods still missing
*/
package pack;

import java.blah.blahdy.BlahBlah;

public class Blah extends SomeClass {
      /* A class implementation comment can go here. */

      /** classVar1 documentation comment */
      public static int classVar1;

      /**
       * classVar2 documentation comment that happens to be more than
       * one line long
       */
      private static Object classVar2;

      /** instanceVar1 documentation comment */
      public Object instanceVar1;

      /** instanceVar2 documentation comment */
      protected int instanceVar2;

      /** instanceVar3 documentation comment */
      private Object[] instanceVar3;

      /**
       * ...method Blah documentation comment...
       */
      public Blah() {
            // ...implementation goes here...
      }

      /**
       * ...method doSomethingElse documentation comment...
       *
       * @param someParam
       *             description
       */
      public void doSomethingElse(Object someParam) {
            // ...implementation goes here...
      }
}
```