Guideline Java Code Convention SBB IT v4

Update March 2019

This Java Code Convention replaces the Java Code Convention v3.0 starting with April 2019. We now only maintain the Java Code Conventions ourselves, if there is no meaningful counterpart in the Google Java Style Guide.
This decision was made in the beginning of 2019.

The *Java Code Convention SBB IT* is maintained by the practice group PG Code Quality. Inputs to these Java Code Conventions can be brought in via this practice group.

Document	
Responsible:	Practice group PG Code Quality
Status:	approved
Version:	4.6
Last Change:	30 Sep 2020
Copyright / Licence:	This document is protected by copyright. Any commercial use requires a prior, explicit approval of the SBB AG. External contents of the Google Java Style Guide have the following licence: https://creativecommons.org/licenses/by/3.0/

Change control (has to be filled out during the edition of the document)						
Version	**Date**	**Change**	**Supervision**	**Approval**	**Executive organization**	**Comments / Type of change**
4.0	26 Feb 2019				Practice group PG Code Quality	Refresh of the Java Code Conventions based on Google Java Style Guide.
4.1	12 Mar 2019				Practice group PG Code Quality	Extracted some parts as best practices in a sub document (not essential part of Code Convention)
4.2	15 Mar 2019	Adigüzel Zafer (IT-SWE-CC1-JV8)	Cotting Nicolas (IT-SWE-CC1-JV3)		Practice group PG Code Quality	Translation into English
4.3	20 Mar 2019	Adigüzel Zafer (IT-SWE-CC1-JV8)			Practice group PG Code Quality	Some corrections
4.4	25 Mar 2019	Adigüzel Zafer (IT-SWE-CC1-JV8)	Cotting Nicolas (IT-SWE-CC1-JV3)	Practice group PG Code Quality	Practice group PG Code Quality	Translation of best practices and fix references
4.5	28 May 2019	Adigüzel Zafer (IT-SWE-CC1-JV8)			Practice group PG Code Quality	Expand function for code blocks
4.6	30 Sep 2020	Adigüzel Zafer (IT-SWE-CC1-JV8)			Practice group PG Code Quality	Translate a sentence and move to new CoP SE space.

Table of content

General

This document describes the binding Java Code Conventions (Java Programming Guidelines) of SBB IT.

Purpose of the Document

This document sets out basic guidelines for writing Java code.
In addition to purely formal and stylistic criteria, these guidelines also set limit values for length and complexity.

Reason / Purpose

SBB IT's Java Code Conventions seek to increase the maintainability, efficiency and reliability of software products.
In addition, they provide a basis for the static measurement of internal code quality by defining potential measurement criteria.

Scope

These conventions apply to all software projects of SBB IT, which are developed using the Java programming language. The conventions therefore apply to JEE applications (Java EE) as well as to Java SE applications, which are based, for example, on Spring Boot (ESTA Cloud). The scope also includes externally implemented software products on behalf of SBB IT, which will be maintained by SBB IT after realization.

This document replaces all other conventions (previous versions of the Java Code Conventions as well as analog documents) and is binding for new projects. Project- or product-specific conventions can extend these conventions, but not override them. This convention regulates the generally valid minimum.

Verification of compliance

Adherence to the specification can be verified by means of code reviews by experts using the following criteria.

- Source is compliant (visual review of source code by review)
- Static code analysis (sonar)
- Use of the predetermined formatting

Must, Should, Can

There is no explicit distinction between must, should and may in the Google Java Style Guide. Items that differ and extend the guide are identified by the following patterns in this document:

- **Must:** Always compulsory.
- **Should:** Deviation with justification (e.g. comment) possible.
- **Can:** Information, recommendation or option.

Content of the specification

Unless otherwise defined in this document, the recommendations issued by the Google Java Style Guide apply. If the requirements of the *Java Code Convention SBB IT* conflict with those of Google, then the Java Code Conventions SBB IT apply.

Extended recommendations (Best Practices) are written in a separate document (Best Practices for "Java Code Conventions SBB v4") and thus not an integral part of the *Java Code Convention SBB IT*.

Length, Complexity and Readability

Inner Length

Should

Length in Java statements (NCSS, see definitions in the appendix).

Inner Length	Max amount NCSS
Methods	100
Classes	1500
Files	1500
Length of lambdas and anonymous inner classes	20

Line Length

Must

- The character length of 100 suggested by Google Java Style Guide (4.4 Column limits) may be exceeded.
- The line length must not exceed 200 characters.

Number of Parameters

Should

A method should have a maximum of 7 parameters.

Complexity

Should

- The cyclomatic complexity of a method after McCabe [4] should not exceed 15.

- The number of possible paths through a method (NPath [5]) should not exceed 300.

Coupling

Should

Classes should not depend on too many other classes. The maximum number of dependent classes should not exceed 20.

Code block 1

```
class Foo {                          // Noncompliant - Foo depends on too many classes: T1, T2, T3, T4, T5, T6
and T7
  T1 a1;                             // Foo is coupled to T1
  T2 a2;                             // Foo is coupled to T2
  T3 a3;                             // Foo is coupled to T3

  public T4 compute(T5 a, T6 b) {  // Foo is coupled to T4, T5 and T6
    T7 result = a.getResult(b);    // Foo is coupled to T7
    return result;
  }

  public static class Bar {        // Compliant - Bar depends on 2 classes: T8 and T9
    T8 a8;
    T9 a9;
  }
}
```

Magic Numbers

Must

"Magic Numbers" are not permitted. Exceptions: -1, 0, 1 and 2

```
day = (3 + numberOfDays) % 7;    // Non compliant

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

day = (WEDNESDAY + numberOfDays) % DAYS_IN_WEEK;  // Compliant.
```

Assignment in subexpressions

Must

Due to poor readability and side effects, no assignment is allowed in subexpressions. Excluded are assignments in while statements.

```
// Non compliant
doSomething(i = 42);

/////////////////

// Compliant
i = 42;
doSomething(i);

// or
int c;
while ((c = in.read()) != -1) {
  out.write(c);
}
```

Imports

Must

- Instructions in Google Java Style Guide: Import Statements
- No import statement for java.lang. This package will be imported automatically.
- Only import classes that are actually used in the Java file (no longer needed import statements have to be removed).

Comments

Must

- Instructions in Google Java Style Guide: Comments and JavaDoc
- Comment only as much as necessary, but consistently. Where clarity can be achieved through naming, this is preferable to a comment.

Language

Must

Javadoc and comments are written in German or English. The decision whether German or English is made by the project

Can

Umlauts in comments are allowed.

Comments in derived Classes

Must

Primarily the interface or the super class is commented. Copying comments has to be avoided.

Can

Additional features or peculiarities in the derived class or implementation can be explained.

File and Class Header

Must

Each class file must have a file header with copyright (see code block):

Can

The class header can be extended with comments, if these add real value or are elementary for the understanding of the class. Basically, comments should only reflect meaningful contextual information.

We advise against the following information, as the use of GIT fully covers this requirement.

- @author <uXXXXXX> the author
- @since<Project version or creation date dd.mm.yyyy> the project version
- @version<unique versioning of the file> the actual unique version of the file

```
/*
 * Copyright (C) Schweizerische Bundesbahnen SBB, <year, 4 digits>.
 */
package ch.sbb.service.zug;

/**
 * If necessary: Class description.
 */
public class ZugService {
}
```

Documentation

The following describes the minimum scope of JavaDoc for each item. Basically, the statements from Google Java Style Guide JavaDoc apply.

Public Interface and Public Class

Must

Definition of the purpose, benefits and functional scope of the interface / class.

Public and Protected Method

Must

- Purpose of the method, unless it is clear from the name.
- Where there are such: Hints to special preconditions and side effects.
    - Example: method `calculateSum()`: Computes the sum of the values and temporarily stores the result.
- Admissibility of `null` as parameter and return value (this comment can be replaced by annotations).
- Multithreading capability or correct synchronization, especially if it impacts performance and scalability (concurrency constraint). If only parts of the API are intended for concurrent use, this must be explicitly marked.
- Public getter and setter methods of JavaBeans must not be commented.

Public Constants

Must

Definition of semantics, if not clearly stated in the name.

Packages

Can

- If required in the file `package-info.java`.
- Purpose of the package (what does it contain?).

TODO / FIXME

In the code, TODO and FIXME comments indicate places in the source that are not or not fully implemented (TODO) or incorrect (FIXME).

Must

- FIXME comments must not occur in the productive code. Instead of writing FIXME comments, the code should be fixed immediately if possible. If this is not possible, the problem must be described. FIXME comments must be labeled with U or E number and date and, if possible, have a reference to a JIRA issue.
- TODO comments must be tagged with the U number, date and, if possible, have a reference to a JIRA issue. TODO should be fixed whenever possible within a sprint.

```
public void doSomething() {
// TODO U215021/19.08.2015: Details of implementation have to be clarified first (JIRA-1234).
}

public int divide(int numerator, int denominator) {
 return numerator / denominator; // FIXME U215021/19.08.2015: denominator can be null. Clarify, how the
application should behave (JIRA-3210).
}
```

Documentation of Special Cases

Must

In the following cases, suitable documentation within the code should be aimed for:

- Unexpected or particularly special or complex solutions
- Workarounds for technical debts of other components
- Special treatments and the like implemented during maintenance or bug fixing have to be commented accordingly in the code at the affected place:
    - Why was this solution chosen?
    - What are the consequences without this solution?
    - Possibly reference to the unique name of the bug

Switch statement fall-through

Must

A `case` in a `switch`, which is not terminated by a `break`, has to be commented (see Google Java Style Guide: Switch statement fall-through)

Declarations

A Declaration per line

Must

- According to Google Java Style Guide (One statement per line) only use one declaration per line. This is clearer and facilitates documentation.

```
int levelInTree = 0, tableSize = 10; // non compliant, two declarations in one line
int levelInTree = 0; // level of the binary tree starting from root
int tableSize = 10;
```

Collection API

Must

The Collection API [2] introduced with Java 1.2 must always be used unless an external API requires the older Collection API (before Java 1.2).

- The class `Vector` has to be replaced with `ArrayList`.
- `HashMap` should be used instead of the `Hashtable` class.
- Instead of `Enumeration`, use the class `Iterator`.

The interface must be used for the declaration. If necessary, the implementation can be used for the declaration. The type (Generics [3]) has to be declared if possible.

```
final ArrayList<String> foo = new ArrayList<String>(42); // non compliant, no interface
final List foo = new ArrayList(42); // non compliant, no type
final List<String> foo = new ArrayList<>(42); // compliant
final Set<Integer> set = new HashSet<>();
// compliant - if addFirst etc. are needed
final LinkedList<String> lili = new LinkedList<>();
lili.addFirst("bla");
```

Collections and maps that are declared `final` can still be changed. Where such changes are not intended, the collection or map must be protected by means of `Collections.unmodifiable*` etc.

```
final List<Integer> items = Collections.<Integer>unmodifiableList(Arrays.asList(0,1,2,3));
```

Annotations

Must

Instructions in Google Java Style Guide: Annotations

Can

At certain points, you can deviate from the rule that every annotation must be placed on a new line. This e.g. can be meaningful when a variety of technically related annotations (e.g., Lombok annotations on a data class) are used and readability would suffer from the resulting new lines:

```
@Getter @Setter @NoArgsConstructor @AllArgsConstructor @ToString(callSuper = true) @Builder
public class VehicleDTO {
        ...
}
```

Formatting

Indentation

Must

Indentation takes place via blocks of 4 spaces. No tabs can be used, the indentation takes place exclusively via spaces.

Switch statement

Must

Instructions in Google Java Style Guide: Switch

Naming conventions

General conventions

Must

- Instructions in Google Java Style Guide Identifier Names and Specific Identifier Names
- For enums and annotations, the conventions defined for class names apply.
- Generic type parameters consist of exactly one capital letter. (ojt)
- The dollar sign ($) is not allowed in variable, method, class and interface names.
- Use camel case to separate name components (e.g., ZugManagerImpl). The use of the underscore "_" is permitted only in the names of constants and in unit test classes.
    - The basis of the rule are the instructions in Google Java Style Guide: Camel Case

Package names

Must

Packages must begin with `ch.sbb` or `ch.voev` for VöV-projects.

Meaningfulness

Should

You should use speaking names. Wherever possible, speaking names are preferable to additional comments.

```
/**
 * Returns the size in the measure of meters.
 *
 * @return size in meters
 */
public int size() { // good
  <return the size somehow>;
}
public int sizeInMeters() { // better
  <return the size somehow>;
}
```

Uniqueness

Should

Names should help to uniquely identify classes, interfaces, methods, variables, and constants.

Classes and Interfaces

Must

- Interface and implementing classes may not use the same name.
- Superclass and subclass may not use the same names.
- Methods of inner classes may not overload methods of the outer class. If this can not be avoided, the call has to be made explicitly with `<inner class>.this.<Method>` or `<inner class>.super.<Method>`.

Methods

Must

The names of methods in a class may not only be different in their case.

Exceptions

Must

Class names ending in "Exception" must extend `Exception` or one of its subclasses.

Language

Must

- All IT terms are in English.
- Technical terms are German or English (decision of the specific project)

```
class PresentationHelper { // IT term
}
class Fahrzeug { // technical term
}
class ZugNotFoundException extends Exception { // Mixed form for technical term Zug
}
class Foo {
  private int size; // IT term
  private int anzahlFahrzeuge; // technical term
  public final void setSize(final int size) { // IT term
    this.size = size;
  }
  public final int getAnzahlFahrzeuge() { // technical term
    return anzahlFahrzeuge;
  }
}
```

Non-ASCII Characters

Must

Outside of comments and string literals, only ASCII characters [7] are allowed, umlauts must not be used.

References: Google Java Style Guide 2.3 Google Java Style Guide 5.1

Encoding

Must

- Instructions in Google Java Style Guide: File encoding

Exception handling

Exceptions only for Special Cases

Must

Control flow via exceptions is not allowed. See also "Use exceptions for exceptional conditions" [6] Item 57 (page 241ff).

```
// Example by Joshua Bloch - non compliant, exception used as control flow.
try {
  int i = 0;
  while(true) {
    range[i++].climb();
  }
} catch(ArrayIndexOutOfBoundsException e) {
}
```

Treat Exceptions as late as possible

Should

Exceptions should only be treated (`catch`), where they can or must be processed in a functional and technically meaningful way (e.g., at the border between layers). Until then, they are to be submitted using the `throws` declaration (as required). Conversely, exceptions are similarly as bound to the encapsulation of layers or technical functionality as the rest of the implementation.

See also [6] Item 61 (page 250).

Don't swallow exceptions (catch)

Must

- Instructions in Google Java Style Guide: Caught Exceptions
- Each exception must be dealt with (rethrow or react accordingly and log). If this does not make sense (i.e. the exception should be swallowed), this decision must be commented.

Don't lose the context

Should

The stack trace should not be lost during error handling. It should be logged and passed on to the exception while wrapping the exception, if possible.

```
// Example without logging
void bar() throws FrameworkException {
  try {
    <do something>;
  }
  catch (SomeException se) {
    // Non compliant, stack trace is lost
    throw new FrameworkException(se.getMessage());
  }
  catch (OutOfLuckException ole) {
    // Compliant
    throw new FrameworkException(ole);
  }
}
```

Don't throw Exceptions in finally-blocks

Must

Exceptions must not be thrown in finally-blocks because they hide the original exception.

```
try {
  /* Some work which ends up throwing an exception */
  throw new IllegalArgumentException();
} finally {
  /* Clean up */
  throw new RuntimeException(); // Non compliant - will mask the IllegalArgumentException
}

////////////////////////

try {
  /* some work which ends up throwing an exception */
  throw new IllegalArgumentException();
} finally {
  /* clean up */               // Compliant
}
```

Throw specific Exceptions

Must

Don't throw `Throwable` or `Exception` directly. Instead use subclasses of them.

```
public void bar() {
  throw new Exception(); // Non compliant, use a specific exception
}
```

Don't throw Errors

Must

`Errors` (see term definitions) may not be thrown (this is reserved for the JVM).

Don't throw NullPointerException

Should

`NullPointerException` should not be thrown (this is reserved for the JDK).

Don't catch Errors

Should

Don't catch `Error`, specific `Errors`, and `Throwable`. Instead of *catch (Throwable t)* the clause *catch (Exception ex)* or *catch(<XYException> ex)* should be used.

Error may not be extended

Must

The `java.lang.Error` class must not be extended. `Error` and its subclasses should only be thrown by the JVM.

Visibility and Validity

Visibility of instance and class variables

Must

Instance and class variables that are not constants must be `private`. Only instance and class variables that are `static final` may be, `public`, `protected`, or package-private (without an access qualifier).

```
public class Foo {
  public String barMutPub; // Non compliant, public instance variable and mutable
  protected String barMutProt; // Non compliant, protected instance variable and mutable
  String barMutPackPriv; // Non compliant, package-visible instance variable and mutable
  public final String barImutPub; // Non compliant, no constant (static final)
  private String barMutPriv; // Compliant
  private final String barImutPriv; // Compliant
  public static final String BAR_STAT_PUB = "abc"; // Compliant, public constant
  private static final String BAR_STAT_PRIV = "abc"; // Compliant, private constant

  <constructor>;

  public final String getBarMutPriv() {
    return barMutPriv;
  }
  public final void setBarMutPriv(final String barMutPriv) {
    this.barMutPriv = barMutPriv;
  }
}
```

Reflection

Must

Changing visibility at runtime using reflection is not allowed.
Exception: Test classes.

Validity of variables

Must

The scope of variables should be as small as possible. When using variables, the following rules must be observed:

- Variables and fields always have the smallest possible scope.
- Wherever possible, local variables are preferable to instance variables.
- Wherever possible, instance variables are preferable to class variables (`static`).
- Local variables have to be declared in the smallest block possible. For example, within a loop if the variable is not needed outside the loop.

Constants

Must

Constants have to be declared `static final`. All fields whose values are initialized hard-coded and never change are constants. The correct naming has to be considered (see Google Java Style Guide Constant Names).

Interfaces as declaration type

Should

Interfaces or abstract classes and not the concrete implementation should be used as the declaration type (on the left side for return values).
This applies to all classes and interfaces, both in external (collection framework) and internal APIs. To realize this pattern, interfaces can be created for your own publicly accessible classes (except for Domain Objects and JavaBeans).

```
public class Bar {
  // Non compliant, if FooImpl is not needed explicitly.
  private FooImpl boese = new FooImpl();
  private Foo gut = new FooImpl(); // Compliant, usual case

  public FooImpl createFoo1() { // Non compliant
    return new FooImpl();
  }


  public Foo createFoo2() { // Compliant
    return new FooImpl();
  }
}
```

Classes without instances

Must

Helper classes with exclusively static methods must be declared `final` and define a `private` (i.e. hidden) default constructor.

```
public final class MyHelperClass {
  private MyHelperClass() { // no instances
  }
  public static <Type> MyHelperMethod(<Param>) {
    <do something>;
  }
}
```

Ineffective Code

Unused Variables and Methods

Must

Private instance and class variables, local variables, and methods must be used at least once or be required for formal reasons. Private instance and class variables that are not read or written at least once and are not required for formal reasons are superfluous and must be deleted.

Unused Parameters

Should

Parameters should be used at least once in the method to which they are passed. Unnecessary parameters should be removed. Exceptions are parameters that are specified by an external or general API.

Commented out Code

Must

Commented out code is prohibited. Unused or temporarily unused code must be deleted. If necessary, source code management serves for the recovery.
Excluded from this is code in comments, which serves explicitly and clearly for documentation purposes.

Empty Blocks

Should

Empty blocks (if, else, for, while, switch, try / catch / finally block without statements between the curly braces) should be avoided. If this can not be prevented, at least one comment should be included as to why no statement should be processed.

```
if (<condition>) { // Non compliant, empty block
}
else {
  <do something>;
}
if (!<condition>) { // Compliant
  <do something>;
}
```

No unreachable Code

Must

Unreachable code must be omitted.

```
if (false) { // unreachable code
  <do something>;
}
```

Ineffective if-statements

Must

If-statements, which are always `true`, must be omitted.

```
if (true) { // Ineffective if-statement
  <do something>;
}
```

Useless Catches

Must

Only for exceptions that can actually be thrown in the context in question, catch statements may exist.

Unnecessary Catch/Throw

Must

Unnecessary *catch* and *throw* of exceptions have to be avoided.

```
void bar() throws SomeException {
  try {
    <do something>;
  }
  catch (SomeException se) {
    throw se; // don't just rethrow an exception
  }
}
```

Appendix

Terms and Abbreviations

Term / Abbreviation	Definition
NCSS	Non commented source statements: Number of statements (Java instructions). A formatting-neutral algorithm to measure the LOC.
LOC	Lines of code: Code lines without blank or comment lines.
Errors	*java.lang.Error* and all derived classes or their instances
Java Assertions	A statement provided by the Java programming language with the keyword `assert`.

Table 1: Terms and Abbreviations

References

Nr.	Link
[1]	Oracle Java Turorials https://docs.oracle.com/javase/tutorial/
[2]	The Java Tutorials, Trail: Collections. Josh Bloch, Sun Tutorial (online), 1995. http://download.oracle.com/javase/tutorial/collections/index.html
[3]	Generics in the Java Programming Language https://docs.oracle.com/javase/tutorial/extra/generics/index.html
[4]	A Complexity Measure, Thomas J. McCabe, IEEE Transactions on Software Engineering Dezember 1976. http://www.literateprogramming.com/mccabe.pdf
[5]	NPATH: a measure of execution path complexity and its applications. Brian A. Nejmeh, Communications of the ACM, 1.2.1988. https://www.semanticscholar.org/paper/Npath%3A-A-Measure-of-Execution-Path-Complexity-and-Nejmeh/ca83e18eabbbc01d8e86897688f2251cc5b8eabe
[6]	Effective Java Second Edition, Joshua Bloch. Prentice Hall, 2 edition, 28.5.2008. ISBN: 978-0321356680
[7]	ASCII format for Network Interchange, Vint Cerf, 10.1969. http://tools.ietf.org/html/rfc20

Table 2: References