**Introduction**

Software development is a fundamental part of solving problems in everyday business processes. In order to come up with standard coding techniques, the process requires well set procedures which should be followed to accomplish the intended goals. In this documentation we will list down all the set rules that are accepted in our company. They will allow every member of the team or interested parties to understand the code, make it more readable, neat and to avoid errors associated with different styles of writing code for people working on the project as well as making it easy to edit it in the future.

**Purpose of the document**

This document defines the standard requirements for developing source code. The document is included in our organisation’s standards that define the requirements for the development of source code in programming languages.   
  
**The purpose of the coding standard:**

* simplify software support through unification
* ensure the possibility of support and development of the software product by its original authors by improving the readability of the source code
* provide the ability to work with the source code (when it is sold as part of a software product) to the buyer's specialists.

The document defines the requirements for the development of Java source code, incl. Java Server Pages (JSP) code requirements.   
  
The standard does not define the principles of source code design, specific development tools, as well as methods of compiling, assembling projects and other technological operations carried out in the software development process.

**Links**

In preparing the document, recommendations from Sun Microsystems, Inc. were used. on the design of Java-code published as follows:-

* <http://java.sun.com/docs/codeconv/>
* <https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
* <https://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html#16712>

Terms and Definitions

* Java source files are files that contain Java class definitions of Java classes.
* JSP files are files containing code in JSP format.
* Java source code — Java source and JSP files.

**Source code placement**

All developed source code of the project is placed in a single repository that supports version control, multi-user work, authentication, authorization, logging of operations and the ability to access via the Internet. For example, GitHub, Microsoft Visual Team Foundation Server, SVN, CVS etc.   
  
The choice of a specific means for storing the source code is determined by the project manager in accordance with the features of the project.

**Formatting Java source files**

File organization

* Components of the source code are separated by blank lines and additional comments.
* Files over 2000 lines in size should preferably be split into several separate files.
* Each Java source file can contain only one public class or interface. Public class or interface must be defined first in the source java-file.

The source Java file should consist of the following components:

* Initial comment
* package and import instructions
* Class definitions and interfaces

### 

### Initial comment

All system source files should begin with a commentary containing copyright information: i.e.  
*/ \* \* Copyright 2019 MSikwese Ltd. All rights reserved*.

### Package and import instructions next to the description of the file, without a space, the package declaration follows, then after the space, the imported classes or interfaces should be declared. Example:  *package com.msikwese.deasp.services;*

*import com.msikwese.deasp.exceptions.ResourceNotFoundException;*

*import com.msikwese.deasp.models.Customer;*

The import order should be divided into the following groups:   
java, javax, org, net, com, etc. There should be one space between each import group, within the group, imported classes should be ordered alphabetically. The use of the \* symbol in import instructions is not allowed (each imported class or interface is written in a separate line). Example:  
*package com.msikwese.deasp.services;*

*import com.msikwese.deasp.exceptions.ResourceNotFoundException;*

*import java.util.Date;   
  
import java.util.ResourceBundle;   
import javax.swing.DefaultListModel;   
  
import javax.swing.JList;   
import org.apache.commons.lang.StringUtils;*

**Class and Interface Definitions**

The section contains definitions of classes and interfaces in Java and comments to them.   
Before the name of the class or interface there should be a comment containing information about the author and the version number of the file, then without a space, the definition of the class or interface. Example:   
*/ \*\*   
  
\* @author Martha Sikwese   
  
\* @version 0.01 Revision: 1.11   
  
\* /*

*public class Blah extends SomeClass {*/ \* A class implementation comment can go here. \* / */ \*\**classVar1 documentation comment*\* /   
  
public static int classVar1;   
  
public Blah () {*// ... implementation goes here ..*}*  
Full example

/ \*   
  
\* Copyright 2019 MSikwese Ltd. All rights reserved.

\* /   
  
package com.msikwese.deasp.services;

import com.msikwese.deasp.exceptions.ResourceNotFoundException;

import java.util.Date;   
  
import java.util.ResourceBundle;   
import javax.swing.DefaultListModel;   
  
import javax.swing.JList;   
import org.apache.commons.lang.StringUtils;   
  
import org.mozilla.javascript.Function; 

* / \*\*   
    
  \* @author Martha Sikwese   
    
  \* @version 0.01 Revision: 1.11   
    
  \* /   
    
  public class Blah extends SomeClass {  
    
  / \* A class implementation comment can go here. \* /  
  / \*\* classVar1 documentation comment \* /  
    
  public static int classVar1;   
    
  public Blah () { // ... implementation goes here .. }

**Indentation**

* Indents, line length, line breaks
* Indents should be strictly 4 spaces (not a tab).
* The length of the string should not exceed 80 characters.
* If the length of the expression exceeds the length of the string, then it is necessary to split it into several lines according to the following rules:
  1. post-comma transfer;
  2. transfer to the operator;
  3. 8 spaces must be indented to indicate the second line of the split expression. Subsequent lines are aligned on the second line or new 8 spaces are added to indicate nesting.

int result = function1 (longExpression1,

function2 (longExpression2,

longExpression3));

**Line length and hyphenation**

The length of the lines of the original Java file should not exceed 120 characters.   
If the expression cannot fit into one line, then it should be divided into several lines in accordance with the following principles:

* The line can be interrupted after the comma
* The string may be interrupted before the operator

When transferring the expression, the second line must have 8 spaces indented with respect to the first, all the following lines must be aligned with the same level as the second line of the expression. For example:   
  
// CONVENTIONAL INDENTATION   
  
someMethod (int anArg, Object anotherArg, yetAnotherArg String,   
  
Object andStillAnother) {   
  
...   
  
}   
  
// INDENT 8 TO AVOID VERY SPACES DEEP INDENTS   
  
private static synchronized horkingLongMethodName (int anArg,   
  
Object anotherArg, yetAnotherArg String,   
  
Object andStillAnother) {   
  
...   
  
}

**Blank lines**

Blank lines should be used to divide a file into logical parts.   
Two consecutive blank lines should be used in the following cases:

* Between sections of the source file;
* Between class and interface definitions.

One blank line should be used in the following cases:

* Between methods
* Before block and single-line comments
* Between the logical parts inside the method to increase its readability

**Comments in Java source files**

Java source file can contain two types of comments.

* Comments like C ++ and defined with / \* ... \* / or //
* Comments for the javadoc utility, defined as / \*\* ... \* /

Comments like C ++ should be used if they are only supposed to be read when directly viewing the source code.   
Javadoc comments are used to automatically build documentation on the source code. They are used both for direct viewing of the document and as descriptions of classes, interfaces, methods and attributes in the documentation automatically generated by the javadoc utility.   
  
Each public or protected class, interface, method, and attribute must contain a comment for the javadoc utility to build documentation.   
The need to include additional comments is determined by the developer or project manager.   
Comments should make it easier to understand the source code.

**Source Code Documentation**

Using the javadoc utility, the source java code should automatically generate documentation in HTML format.   
The source code must contain enough comments to describe in the javadoc utility documentation of each public and protected class, interface, method and attribute.

**Naming**

General naming rules

* Naming should simplify the understanding of the program code and facilitate its reading.
* Names of packages, classes, methods and attributes and constants must match the functionality they implement.

**File naming**

Java source files must have the same name as the class or interface contained in them and have the extension .java, in accordance with the rules of the Java language.   
  
JSP-files must have the extension .jsp and the name consisting only of Latin letters, numbers and underscores without spaces. The name of the JSP file must correspond to the operations it performs.

**Package naming**

Package names must begin with package com.msikwese. and further contain a unique name consisting only of Latin lowercase letters (ASCII 97-122) and points as a separator of sub packages. Example: *package com.msikwese.deasp.services*;

**Naming classes and interfaces**

Classes are referred to as mixed case letters. Each word that constitutes the name of the class begins with an uppercase letter, the remaining letters of the word are written in lowercase. For the naming of classes should use the correct words of the English language. The use of abbreviations other than generally accepted is not allowed. In abbreviations, uppercase should be used only for the first letter of the abbreviation, the other letters of the abbreviation are written in lowercase. Example: *public class HttpUserRepository { ...*

**Method naming**

Methods must be named according to the same principle as classes, but all method and attribute names must begin with a lower case letter. Example:

*public void doSomething() {*// ... implementation goes here ...*}*  
**Variable naming**

Attributes and variables are named according to the principle as the methods, with the exception of temporary variables, which are allowed to be called one letter.   
It is not allowed to start variable names with “\_” and “$” characters.

**Naming Constants**

Constant names must consist of uppercase letters and the \_ character as a line delimiter. The names of constants should not be used not common abbreviations. Example:   
*static final int MAX\_WIDTH = 999;*

**Declarations**

Number of declarations per line

It is not allowed to include several declarations in one line.  An example of valid declarations:

*String managerName;   
int code*;

An example of invalid declarations:   
*String managerName; int code; // NOT ACCEPTABLE!*

**Variable initialization**

Variables should be initialized when they are declared in cases where the initial value of the variable does not depend on the calculations.

**Location of Declarations**

Declarations should be located at the beginning of the program blocks. Example:   
*void myMethod () {   
int int1 = 0; // beginning of method block   
if (condition) {   
int int2 = 0; // beginning of "if" block   
...   
}   
}*

It is not allowed to use variable names in nested blocks that match the names of external blocks. Example:   
*int count;   
...   
myMethod () {   
if (condition) {   
int count = 0; // NOT ACCEPTABLE!*   
*...   
}   
...   
}*

**Class and interface declarations**

The following principles should be observed in class and interface declarations:

* The symbol "{" should be at the end of the first line of the declaration
* The “}” character must be the only character in the last line of the declaration, with the exception of empty declarations that can be written entirely on one line
* There should be no spaces between the name of the methods and the “(”) symbol that starts the list of parameters.

**Arrays**

* When initializing an array, there are no spaces after the initial brace and before the final brace. Example:   
    
  *int [] array1 = new int [] {1, 2, 3}; int [] array2 = new int [] {1, 2, 3}; // NOT ACCEPTABLE!*

**Instructions**

Simple instructions

* Each line must contain no more than one instruction. An example of proper execution of instructions:   
  *argv ++; // Correct   
  argc--; // Correct*   
    
  Example of incorrect execution of instructions:   
  *argv ++; argc--; // NOT ACCEPTABLE!*

**Compound instructions**

Compound instructions are instructions consisting of a set of instructions enclosed between "{" and "}" characters.   
  
The following principles should be observed when making composite instructions

* Nested instructions should be indented with respect to external instructions
* The symbol "{", which begins nested instructions, must be at the end of the first line of the external instruction
* The symbols "{" and "}" should frame any sequence of nested instructions, including and consisting of one instruction
* The terminating "}" character of the compound instruction must be located at the level of the first character of the first line of the compound instruction.

Example:   
*if (condition) {   
managerName = username [count];   
count ++;   
} else {   
return false;   
}*

**If, if-else, if else-if else instructions**

The if-else category instructions should be formatted as follows:   
*if (*condition*) {// The presence of a pair of parentheses is mandatory (even if one statement)*statements*;   
}   
if (*condition*) {// The presence of a pair of parentheses is mandatory (even if one statement)*

statements*;   
} else {*statements*;   
}   
if (*condition*) {// The presence of a pair of parentheses is mandatory (even if one statement)*statements*;   
  
} else if (*condition*) {*statements*;   
  
} else {*statements*;   
  
}*

**For instruction**

The for statement is formatted as follows:   
*for ( initialization ; condition ; update ) {*statements*;   
  
}*

For an empty for statement, use the form:   
*for (*initialization*;*condition*;*update*);*

**While statement**

The while statement is structured as follows:   
*while (*condition*) {// The presence of a pair of parentheses is mandatory (even if one statement)*statements*;   
  
}*  
**For an empty while statement, use the form:**while ( condition );

**Do-while instruction**

The for statement is issued as follows:   
*do {*statements*;   
  
} while (*condition*);*

**Switch instruction**

The switch statement is issued as follows:   
*switch (*condition*) {   
  
case ABC:*statements*;   
  
/ \* falls through \* /   
case DEF:*statements*;   
  
break;   
case XYZ:*statements*;   
  
break;   
default:*  
statements ;   
  
break;   
  
}

**Try-catch statement**

The try-catch statement is issued as follows:   
*try {*statements*;   
  
} catch (ExceptionClass e) {*statements*;   
  
}*

In the case of finally using :  
*try {*statements*;   
  
} catch (ExceptionClass e) {*statements*;   
  
} finally {*statements;  *}*

**Error processing**

To handle errors, use the Exceptions mechanism (see **Try-catch statement**)  
Access to methods and variables

Access to class attributes

Do not declare class attributes as public for no particular reason. It is preferable to provide access to attributes indirectly through methods with names beginning with get for reading the attribute value and with set to set the new value.

Access to static methods and variables

Avoid access to static methods and variables through the object name and use the class name to access. Example:   
  
*classMethod (); // OK   
AClass.classMethod (); // OK   
anObject.classMethod (); // NOT POSSIBLE**Constants*

Numeric constants should not be used in direct form, with the exception of –1, 0, and 1 in the for loop counters.

**Assignment to variables**

Assigning multiple variables in one instruction should be avoided. Example:   
*fooBar.fChar = barFoo.lchar = 'c'; // UNDESIRABLE*It is not allowed to use nested assignments. Example:   
*d = (a = b + c) + r; // MUST NOT be written:*

*A good example  
a = b + c;   
d = a + r;*  
  
**Java Server Pages (JSP) Design**

* In JSP, you should not use Java code directly in a JSP file. In place of this, use TagLibrary technology.
* JSP-file should contain only HTML-text and JSP-tags (including tags TagLibrary). All Java code must be implemented as TagLibrary classes and be in separate Java files that are designed in accordance with this standard.
* JSP tags are designed according to the general rules for HTML tags set forth in the HTML Coding Standard.

**Compliance with the requirements of the standard**

* Java code developed by the company should be developed in accordance with this standard.
* Compliance with the requirements of this standard is mandatory for all specialists of the company involved in the development of Java code.
* The Java software code being developed may not comply with this standard only if the customer of the solution being developed has different requirements for the design of the program code. The decision on non-compliance with this standard is made by the project manager.
* Control of conformity of the code being developed to this standard
* The compliance of the developed code with this standard is monitored during the testing process of the developed solution without fail, by specialists testing the solution.

Appendix A. Java Code Example  
  
/ \*   
  
\* Copyright 2005-2006 CJSC Diasoft. All rights reserved.   
  
\* /   
  
package ru.diasoft.blah;   
import java.blah.blahdy.BlahBlah;   
  
/ \*\*   
  
\* @author Andrey Ivanov   
  
\* @version $ Revision: 1.19 $   
  
\* /   
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 the BE \* more than one's line long \* / the private static classVar2 the Object; / \*\* instanceVar1 documentation comment \* / public Object instanceVar1;  
   
/ \*\* instanceVar2 documentation comment \* /   
  
protected int instanceVar2;   
/ \*\* instanceVar3 documentation comment \* /   
  
private Object [] instanceVar3;   
/ \*\*   
  
\* ... constructor Blah documentation comment ...  
  
\* /   
public Blah () { // ... implementation goes here ... } / \*\* \* ... method doSomething documentation comment ... \* / public void doSomething () { // ... implementation goes here ... } / \*\* \* ... method doSomethingElse documentation comment ... \* @param someParam description \* /  
  
  
public void doSomethingElse (Object someParam) { // ... implementation goes here ... } }

Unit – testing is a process in programming which allows to check on the correctness of the individual modules of the source code of the program. The idea is in fact to write tests for each non-trivial function or method.

## Benefits

The purpose of the modular test is to isolate individual parts of the program and show that these parts are functional separately. This type of testing is usually performed by [programmers](https://dic.academic.ru/dic.nsf/ruwiki/44603/11604).

### 

### Encouraging change

Modular testing later allows programmers to spend refactoring being sure that the unit of the system still works correctly (regression testing). It encourages programmers to change the code as quite easy to check that the code works and after the changes.

### Simplify Integration

Unit testing helps to eliminate doubt on the occasion of the individual modules and can be used for the approach to the testing of " bottom up " : first tested the individual parts of the program then the program in general.

### Documenting code

Unit tests can be viewed as a “ living document ” for the class under test . Customers , who do not know how to use this class , can use the unit - test in an example .

### 

### Separation of the interface from implementation

Since some classes may use other classes , testing of a particular class often extends to those associated with it . For example , a class uses a database ; in the course of writing the test programmer discovers , that the test has to interact with the database . This is a mistake because the test is not must go beyond the class boundary . As a result, the developer is abstracted from the compound with the base data and implementing this interface using your own the mock object. This leads to less related code , minimizing dependencies in the system.

## 

## Restrictions

As and any technology tests , modular testing does not let you catch all errors program. In the case it follows from the practical impossibility of tracing all the possible ways of implementation of the program , for the exception of the simplest cases . Besides that , there is testing each of the modules by separately . This means , that the error of integration , system -level functions performed in several modules not to be identified . Besides that , this technology is useless to conduct tests on performance . In this way, modular testing more efficiently when using a combination with other testing methods.

Testing software maintenance - combinatorial task . For example , each possible value of a Boolean variable requires two tests: one on the embodiment TRUE another- for variant FALSE. The result for each line the original code required 3 - 5 rows of test code. To obtain the benefits of the modular test required strictly follow technology testing during the entire process of development of software maintenance . You need to keep not only write about all conducted tests, but and about all the changes to the source code of all modules. For this purpose, the system version control software should be used. In this way , if a later version of the software did not pass the test which was successfully performed earlier , it is simple to check the source code versions and fix the error . Also, you must make sure to constant monitoring and analysis of failed tests . Ignoring this requirement will lead to an avalanche-like increase in unsuccessful test results.

## Unit testing applications

### Extreme Programming

[Extreme programming](https://dic.academic.ru/dic.nsf/ruwiki/44603/8707) assumes the use of automatic unit testing tools as one of the postulates. This toolkit can be either created by a third party ( for example , Boost . Test ), or it can be created by the development team of this application.

In extreme programming , unit tests are used to develop through testing . To do this, the developer before writing the code writes tests that reflect the requirements for the module. Obviously neither one of these tests before writing code to work do not have to. The further process comes down to writing the shortest code satisfying this set of tests.

### Unit Testing Technique

## Tools

All the java program testing in this organisation will be done using JUNIT([JUnit . org](http://junit.org/)) as a recommendation for high level languages