**MTM Standard for Java Source Files**

**Version 3.11**

*August 16, 2014*

A. Frank Ackerman

*Software Engineering  
Montana Tech*

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comment |
| 1.0 | 07/27/08 | Frank Ackerman | Initial Mtech version |
| 2.0 | 06/03/09 | Frank Ackerman | Initial version for SE3250F09 and SE4270F09 |
| 3.0 | 02/01/10 | Frank Ackerman | Full version that includes packages |
| 3.1 | 04/24/10 | Frank Ackerman | Minor comment changes |
| 3.2 | 04/28/10 | Frank Ackerman | Minor changes |
| 3.3 | 08/23/10 | Frank Ackerman | MF and MO suffixes |
| 3.4 | 10/18/10 | Frank Ackerman | Minor changes. Change to class headings and comment on closing brace. CALLING TREE section added to method header. METHODS heading added to class header. |
| 3.5 | 10/25/10 | Frank Ackerman | Minor changes. Remove Application Imports unless there are some. Most of the Mtech applications do not have any. Set ‘\*’ comments to match Eclipse default format. |
| 3.6 | 12/10/10 | Frank Ackerman | Update for class final exercise |
| 3.7 | 07/28/11 | Frank Ackerman | Both bracket styles |
| 3.8 | 08/30/11 | Frank Ackerman | Incorporate javadoc |
| 3.9 | 09/19/11 | Frank Ackerman | Set javadoc comments to Eclipse default |
| 3.10 | 09/24/11 | Frank Ackerman | Normalize MM |
| 3.11 | 08/16/14 | Frank Ackerman | Update for ESOF486F14 |

**Montana Tech Software Engineering Students:**

These Montana Tech Method software engineering standards encapsulate Dr. Ackerman’s decades of experience in the software industry, the IEEE software engineering standards, and many suggestions from various texts. They have gone through many revisions and additions over the last several years. They are part of your software engineering studies so that (1) you may have the experience of developing software to a standard (which you may find you need to do if you take a job that requires high reliability software), and so that (2) you will have the experience of developing high quality software. You are invited to participate in the continuing evolution of these standards by studying them critically and making suggestions for their improvement and correction.

# Purpose

The purpose of this document is to define the Montana Tech Software Engineering standard for Java source files (.java files).

# Introduction

Industry source files tend to have very long lives and to change regularly. Since these files constitute the first level of documentation of the executing modules, it is imperative that source files, as well as being syntactically and semantically correct, be readily understandable by a defined community of knowledgeable professionals. The community addressed by this standard is the students and faculty taking or teaching software engineering classes offered by the Montana Tech CS Department, especially students majoring in software engineering.

The conventions in the "official" Java Code Conventions document (<http://java.sun.com/docs/codeconv/>) are for the most part included in this standard. This standard lays out additional conventions for:

* source file title block
* default package comment
* import headers
* class definitions
  + - class header
      * class constant, object/variable, constructor, and method header blocks
        + method header comment blocks

Method header comment blocks may contain two types of comments: (1) a mandatory javadoc comment and an optional subsequent ordinary comment with the method/function subsections called for in our other MTM source file standards.

mandatory javadoc comment

DESCRIPTION: *methodName)*rest of javadoc explanatory sentence.

javadoc @params sentences if the method has parameters (each @param must be at the beginning of a line)

javadoc @return sentences if the method is not void (@return must be at the beginning of a line)

Optional ordinary comment with the subsections described in section 5.

Not all methods require headers. See above about class headers. If a method does have a header, it must have at least a DESCRIPTION section.

* tagging right braces, and
* naming and tagging variables.

The purpose of these additional conventions is to:

* reduce initial coding errors,
* make it possible to more easily find defects during code inspections,
* to reduce ambiguity in function header NOTES, REQUIREMENTS, DESIGN, TEST CONDITIONS, CORRECTNESS ARGUMENTS, and CALLS
* to facilitate the construction of CORRECTNESS ARGUMENTS.

This version of this standard (version 3.11) applies only to Java application that use only the default package.

This document specifies the requirements that Montana Tech software engineering Java source files should meet to be readily understood by knowledgeable students and faculty taking or teaching Montana Tech software engineering courses. All Java source files created or used in Montana Tech software engineering courses should adhere to this standard.

The MTM software engineering standards will ultimately include source file standards for C, C++, C#, Java, and Python. Although specific language or documentation features make it impossible for these standards to be 100% consistent, wherever possible unnecessary inconsistencies have been eliminated. Where there are inconsistencies these will at some point be noted in the last section of each standard.

# Application

This standard applies to all source files that instructors use as examples and to all source files that students produce as part of a scored assignment.

Two types of .java source files are distinguished:

* those that contain only the application’s main() entry point, and
* those that do not, that is, they just define a class whose objects may be instantiated elsewhere, and whose methods must be invoked elsewhere.

For the first type of .java file the only method in the class should be:

public static void  
 main(String args[]) {  
 ....  
 }//main()

This class should be give the name Main.java for applications to be tested by robot judges, or *ApplicationName*Launch.java for any other application

# .java File For Class Containing main()

The .java file that defines the class that contains the main() method shall have the following parts in the following order:

1. Title block – always present
2. Java package block
3. Class heading comment block
4. Class definition header
5. main() function header block
6. Definition of main() method.
7. Closing brace for class definition
8. Comment for end of source file

The Simple.java file displayed below illustrates some of these items:

|  |
| --- |
| //Project: Simple  //File name: SimpleLaunch.java  //Programmer: Frank Ackerman 8/15/14 initial version  //Attribution: from Gaddis, Starting Out With Java, 1st Ed. |
| //using default package |
| /\*\*  \*Textbook example to show parts of a .java file containing the main()  \*method for an application.  \*/ |
| public  class SimpleLaunch { |
| //-------------  //Member Methods  //------------- |
| /\*\*  \* DESCRIPTION: main  \* Textbook illustrative program.  \*/ |
| public static void  main(String[] args) {  Simple simpleMO = new Simple();  simpleMO.run();  }//main() |
|  |
| }//SimpleLaunch |
| //end SimpleLaunch.java |

Title Block

The title block consists of three Java // comment lines that begin at the left margin. These are the first three lines of the file.

Java package block

This version of this standard (Version 3.11) uses only the default package

Class header comment block

Not all classes require a header. Some Java applications make use of some short, simple classes. Often well-chosen class, attribute, and method names suffice to document the functionality of these classes. In general, interfaces do not require headers.

A class header comment block begins with a blank line. The following is an example:

/\*\*

\* Textbook example to show parts of a .java file.

\*/

Note: This is the format automatically created by Eclipse (a common IDE used for constructing Java programs). This is the format that javadoc expects. Other javadoc items by be included here but are not required by this standard.

Class definition header

In the following schema notice:

* that the class definition header begins immediately after the class header comment
* that class definitions begin in column 1
* the brace that closes the class definition starts in column 1 and is commented with the name of the class.

public class

Simple {

...

}//Simple

main() method heading comment

The main() method header begins with a blank line. The following is an example of a header with all possible sections. A DESCRIPTION section is always required. For a very simple main() function only a DESCRIPTION is required.

/\*\*

\*DESCRIPTION: main

\* *Brief description of this application  
 \*or reference to a* *Simple Program Specification*

\*/

Definition of main() method

The definition of the main() method immediately follows the main() header. Notice

* that method attributes are give on a separate line immediately above the function signature
* the parameters for main() are always String[ ] args
* that tabs are set at 4, 8, 12, ...
* that method code begins at the 3rd tab stop
* that closing right braces are always commented to indicate the item being closed

public static void  
main (String args[]) {

*Method Body*

}//main()

}//SimpleLaunch

//end SimpleLaunch.java

Closing brace for class definition

The closing brace should be preceded by a blank line. The brace includes a  
 //*nameOfClass*

Comment for end of source file

//end *nameOfFile*

# java File for Any Other Class

Title block

Same as for class that contains main().

Java package block

Same as for class that contains main().

Java imports header comment block

A Java imports block begins with a blank line. The following is an example.

//-------------

//Java imports

//-------------

Java imports

Listed one pre line, immediately below the imports header comment block or  
 //none

Application imports header comment block

An application import block begins with a blank line. The following is an example.

//-------------------

//Application imports

//-------------------

Application imports

Listed one pre line, immediately below the imports header comment block or  
 //none

Class header comment block

Same as for class that contains main().

Class definition header

Same as for class that contains main().

Class member constants comment block

A class constants comment block begins with a blank line. The following is an example

//---------------

//Member Constants

//---------------

Class member constants

Listed one pre line immediately below the constants comment block, or //none. Constants are named using upper case letters and the underscore. For example, MAX\_LENGTH.

Class member objects comment block

A class objects comment block begins with a blank line. The following is an example

//--------------

//Member Objects

//--------------

Class member objects

Listed one pre line immediately below the objects comment. Class object/variable names are suffixed with MO. All class objects should be listed here unless they must be dynamically created later in the program.

Class constructors comment block

A class constructors comment block begins with a blank line. The following is an example

//------------------

//Class Constructors

//------------------

Class constructor definitions

Constructors follow the same standards as other methods except

* 1. they must have the same name as the class,
  2. they must be public, and
  3. they cannot have a type (even void).

Each constructor must have a different signature.

Class methods comment block

A class methods comment block begins with a blank line. The following is an example.

//--------------

//Member Methods

//--------------

Class method/function names except for main, constructors, and extended or overwritten functions are suffixed with MM.

Method heading comment block

A method header begins with a blank line. The following is an example of a header with all possible sections. A DESCRIPTION section is always required. For a very simple function only a DESCRIPTION is required.

/\*\*

\* DESCRIPTION: *methodName*

\* *Brief description of this  
 \* application or reference to a* *n SMDS*

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

REQUIREMENTS:

*Application requirements statement or reference to a*

*Simple Module Specification*

TEST CONDITIONS:

*Test conditions or reference to a SMDS*

TEST DATA:

*Reference to test files or reference to a SMDS*

DESIGN:

*main() design or reference to a SMDS*

CORRECTNESS ARGUMENT: *(optional)*

*Correctness Argument or reference to a SMDS*

CALLS:

*A list of all the routines (including the invocation of*

*constructors) that are called from this method*

*in the order in which they appear in the listing.*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Method definition

The definition of a method immediately follows the method header. Notice

* that method attributes are give on a separate line immediately above the method signature
* that tabs are set at 4, 8, 12, ...
* that method code begins at the 3rd tab stop
* that closing right braces are always commented to indicate the item being closed

public static void  
*methodName*(*method parameter suffixed with PR*) {

*Method Body:*

*Code for method leading label linked to DESIGN in the header comment. See Section 6 below*

}//*methodName*()

Definition of other class methods

Other class methods are defined similarly to the main method except these methods will not contain TEST CONDITIONS or TEST DATA sections.

For all methods with parameters (except main()) the names of the parameters are suffixed with PR.

Closing brace for class definition

The closing brace should be preceded by a blank line. The brace includes a  
 //*nameOfClass*

Comment for end of source file

//end *nameOffile*

The .java file below is an example. Notice:

* that attribution is given in the file header since this code was derived from the code cited
* that method code is indented enough to provide space for a 7 character label

# //Project: MadLib

# //File name: MadLib.java

# //Programmer: Frank Ackerman 8/15/14 initial version

# //Attribution: From Keith Vertanen's CSCI135F13 Assignment 0

# //default package

# /\*\*

# \* An example of the standard way of invoking a Java application

# \*/

# public class

# MadLibLaunch {

# /\*\*

# \*DESCRIPTION: main

# \* Invocation requires 4 parameters that

# \* are used in displaying an "Old MacDonald" MadLib:

# \* (1) an adjective

# \* (2) a noun

# \* (3) an animal

# \* (4) a sound (e.g., Moo)

# \*/

# public static void

# main(String[] args) {

# MadLib madLibMO = new MadLib();

# madLibMO.runMM(args[0], args[1], args[2], args[3]);

# }//main()

# 

# }//MadLibLaunch

# //end MadLibLaunch.java

//Project: MadLib

//File name: MadLib.java

//Programmer: Frank Ackerman 8/15/14 initial version

//Attribution: From Keith Vertanen's CSCI135F13 Assignment 0

//default package

/\*\*

\* This class just displays a MadLib

\* for "Old MacConald had a farm"

\*/

public class

MadLib {

//----------------

//Member Constants

//----------------

//none

//--------------

//Member Objects

//--------------

//none

//-------------------

//Member Constructors

//-------------------

//none

//--------------

//Member Methods

//--------------

//displayMadLibMM

//runMM

/\*\*

\* DESCRIPTION - displayMadLibMM

\* A MadLib for "Old MacDonald had a farm"

\*

\* This function takes four arguments that

\* are used in displaying "Old MacDonald" MadLib:

\* (1) an adjective

\* (2) a noun

\* (3) an animal

\* (4) a sound (e.g., Moo)

\*/

public void

displayMadLibMM(String macAdjctvPR, String macNounPR,

String macAnimalPR, String macSoundPR) {

System.out.println(macAdjctvPR+ " Macdonald had a " + macNounPR +

", E-I-E-I-O");

System.out.println("and on that " + macNounPR + " he had a " +

macAnimalPR + ", E-I-E-I-O");

System.out.println("with a " + macSoundPR + " " + macSoundPR + " here");

System.out.println("and a " + macSoundPR + " " + macSoundPR + " there");

System.out.println("here a " + macSoundPR + ", there a " +

macSoundPR + ",");

System.out.println("everywhere a " + macSoundPR + " " + macSoundPR + ",");

System.out.println(macAdjctvPR+ " Macdonald had a " + macNounPR +

", E-I-E-I-O");

return;

}//displayMadLibMM()

/\*\*

\* DESCRIPTION runMM

\* Run application

\*/

public void

runMM(String macAdjctvPR, String macNounPR,

String macAnimalPR, String macSoundPR) {

displayMadLibMM(macAdjctvPR, macNounPR, macAnimalPR, macSoundPR);

return;

}//runMM()

//----------------------

//Member Getters/Setters

//----------------------

//none

}//MadLib

//end MadLib.java

Class getters/setters comment block

A class setters/setters comment block begins with a blank line. The following is an example

//---------------

//Getters/Setters

//---------------

Private class definitions

All the rules given herein apply to private classes defined within a public class except for additional tabs to delineate hierarchy.

# Method Body

A method body is all of the code that appears between the opening left brace in a method definition and the corresponding closing right brace. See section 5 for an example of the context for a method body.

The top level constructs in a method body must be tabbed over 2 stops from the method definition header.

A method body consists of:

1. method constant definitions
2. method variable declarations
3. method code

Method Constant Definitions

Method constant (final) definitions should be listed before any method variables with a blank line separating constant definitions from variable declarations. The rules for constant definitions are the same as the rules given below for variable declarations except constants are named using only upper case letters (and digits) with parts of the name separated by underscores.

Method Variable Declarations

Method variables should all (except for special cases where they must be defined dynamically) be declared at the beginning of a method before any executable statements. The following guidelines apply.

1. Variable types should be listed in alphabetical order.
2. Each variable type should appear only once with all the variables of that type listed alphabetically on separate lines separated by commas (except see the following).
3. Except for unusual situations, each variable declaration is made on a separate line.[[1]](#footnote-1) Variables of the same type that are closely related, for example, are attributes of the same object, may sometimes be on the same line to show this close relationship. For example, you might have  
    float boxHeight, boxWidth, boxLength; //box dimensions  
   on the same line.
4. Except where required for dynamically allocated variables, variables should not be initialized in a method declaration statement. Such initialization is part of the program logic and should be called out in a design statement that has a corresponding label in the code. (But see 6 below.)
5. Object/variable names
   1. should always being with a lower case letter,
   2. should be descriptive of the data they contain, for example, firstPrime
   3. should not be ordinary English words like prime or first,[[2]](#footnote-2)
   4. should use internal capitals to delineate separate words, for example, lastName rather than last\_name,
   5. should, where appropriate, clearly associate the name with the object it is naming, for example, empLstNm for "employee last name", and
   6. should use complete words except for the following abbreviations[[3]](#footnote-3):

|  |  |
| --- | --- |
| **Abbreviation** | **Entry** |
| avg | average |
| coef | coefficient |
| cord | coordinate |
| crrnt | current |
| diff | difference |
| dsply | display |
| elem | element |
| flg | flag |
| frst | first |
| hrs | hours |
| min | minutes |
| ndx | index |
| nmbr | number of (amount) |
| num | Identifying number |
| nxt | next |
| rslt | result |
| scnd | second |

1. that must be initialized when they are declared should be listed separately at the end of the initial group of declaration statements and, when L lables are being used, given the label L*nn* to match a variable initialization statement in the DESIGN. Variable initialization is an important part of the logic of a program.
2. Variables that have no other function except to index through a single for loop may be declared in the for statement.
3. The above rules apply to naming method parameters except:
   1. method parameters should always be suffixed with PR, and
   2. member objects/variables should always be suffixed with MO.

Method Code

### Executable program statements

The statements that make up the body of a program should adhere to the guidelines given below. In addition there are a number of rules for the use and placement of braces. These are covered in a separate section below.

1. Statement blocks that are part of compound statements must be indented one additional tab stop.
2. Statement should not line break when they are printed,[[4]](#footnote-4) i.e., the programmer must explicitly break long statements. Subsequent lines of broken statements are indented two spaces unless alignment of similar parts is being used.
3. Blank lines should be used to group logically related statements together.
4. Most terminating (right) braces should be on a line by themselves at the same indentation level as the statement they terminate. Most terminating braces should be tagged with the type of statement they terminate, for example, }//for. Additional information may be included in the tag where appropriate, for example, }//for each employee.[[5]](#footnote-5)
5. Unary operators appear immediately adjacent to their operand.
6. Binary operators are almost always surrounded by spaces and the factor or term enclosed in parentheses.[[6]](#footnote-6)
7. Use of the special else if construction should be reserved for a generalized switch, that is, multi-way branching statements.
8. It is permissible to use a while(true) construction. When using this construction the first statement in the loop should be an if-break statement.
9. Spaces should be used in parenthesized expression to clarify scope, for example

foo(!((canNmbr == empNmbr) && (canName != empName)))

should be written as

foo( !( (canNmbr == empNmbr) && (canName != empName) ) )

# Rules For Braces

Braces, like parentheses in expressions, should be used liberally. The following rules apply:

1. if blocks
   1. always use braces, even for one line ifs unless the entire construction fits on one line and else it not used.
   2. always use the format:

if (*expression*) {

*first statement controlled by* if

*any additional statements controlled by* if

}//if *optional descriptive comment*

1. else blocks
   1. always use braces, even for one line else
   2. always use the format:

else {

*first statement controlled by* else

*any additional statements controlled by else*

}//else *optional descriptive comment*

1. while blocks
   1. always use braces, even for one line whiles
   2. always use the format:

while (*expression*) {

*first statement controlled by* while

*any additional statements controlled by* whil*e*

}//while *optional descriptive comment*

1. do while blocks
   1. always use braces, even for one line whiles
   2. always use the format:

do {

*first statement controlled by* do

*any additional statements controlled by* do

} while (*expression*);

1. for blocks
   1. always use braces, even for one line fors
   2. always use the format:

*for* (initializing-list; *expression*; *altering-list*) {

*first statement controlled by* fo*r*

*any additional statements controlled by* for

*}//for optional descriptive comment*

* 1. when the expression controlled by a for loop can fit on the same line the block structure need not be used unless there is a need to parallel an else.

1. switch blocks
   1. the case expressions also starts at the same indent level as switch
   2. the body of each case is indented one stop
   3. there is a blank line before each case and a blank line before the closing brace

switch (*integral-expression*) {

case *value-1*:

*statement1;*

*statement2;*

case *value-2*:

*statement3;*

*statement4;*

default:

*statement5;*

}//switch *optional descriptive comment*

Note that switch causes execution to start at a selected case statement (or the default statement) but unless the last statement in a case is a break statement execution continues with the next case.

1. Putting each variable on a line by itself facilitates (a) following the variable by a brief descriptive comment, (b) initializing that variable, (c) changing the type of that variable, (d) organizing variable declarations into meaningful groups. [↑](#footnote-ref-1)
2. Code documentation that is not embedded in source files has a strong tendency of getting lost over time. (Source code files often migrate over many generations of a program. There are files that were created thirty years ago that are still in use today.) This standard emphasizes *in situ* documentation that may include a detailed design in the function heading comment that is directly keyed to the code. These detailed design statements often reference program variables. Since source code is restricted to a single font, names of variables that are English words may lead to confusing and ambiguous English statements. [↑](#footnote-ref-2)
3. This list is expected to grow. [↑](#footnote-ref-3)
4. The length of a non-breaking line varies with font size and the printer. A good font size is 10. At this size most printer will not break an 80 character line. [↑](#footnote-ref-4)
5. Code is often moved and reformatted. Tagging terminating braces is very helpful in re-establishing visual clues to code logic. It also aids readability generally. [↑](#footnote-ref-5)
6. In some special situations where clarity is enhanced binary operators are not surrounded by spaces and parentheses are not used. [↑](#footnote-ref-6)