**JAVA CODING STANDARDS**

# LAYOUT

* The manner in which coding is visually laid out and presented on the page is the most obvious factor affecting readability.
* Coding which is tightly packed on the page is difficult to read. Use blank lines to separate logical groups of code.
* Similarly, a line of asterisks or dashes before a subprogram helps identify and separate the procedure code.
* Use consistent indenting and alignment of nested control structures and statements. In general, indent any statement that is part of or embedded within another statement.
* 3 spaces are used for each level of indentation.
* Line length is 80 columns. Break compound statements over multiple lines.

# COMMENTS

The use of comment statements is crucial in aiding comprehension of the program code.

#### Explaining your logic

Every control structure block (loop, decision, case, sequence block) has a descriptive comment on the preceding line which uses the vocabulary of the problem space to describe what the logic is accomplishing.

Example:

//Does user want to continue?   
if (Reply == 'y')

Example:

// Consider each letter in the name   
for (int Letter = 1; Letter <= NameSize; Letter = Letter + 1)

#### Class header

Each class belongs in a separate source file that must begin with a header block in the format recognized by the javadoc utility. You should include two descriptive tags

* @author – provide the name of the author(s)
* @version – the date the class was written and/or the version

Example:

/\*\*

\* This is the description of what the

\* purpose of the class is.

\*

\* Additional details

\*

\* @author (yourName)

\* @version (the date class was written)

\*

\*/

The header includes class name, an abstract, @author, and @version (including date). These and all other tags are used in creating documentation. The details field contains a general description of the purpose of the program, what it does, and an overview of the solution.

#### Methods

Each method (except for **main -** which you will encounter later) starts with a header in **javadoc** format that clearly explains its purpose and interface. The interface is described using @param and @return tags. There should be a @param tag for each parameter if they exist and a @return tag if there is a return statement.

Example :

/\*\*

\* Returns the full name

\*

\* @param String first name

\* @param String last name

\* @return the full name

\*/

public String getFullName(String firstName, String lastName)

{

return firstName + `` `` + lastName;

}

#### Declarations

Every identifier declaration has an explanatory "side-bar" comment. Comments are aligned.   
Example:

                char reply;      // User's response

                float balance;   // Current account balance

                dayOfWeek today; // Current day of the week

#### Code Annotations

* Sidebar comments, to the right of the statement, are used to explain any difficult or unobvious coding. Comment any statement whose intent is not immediately obvious. Avoid comments that do not significantly enhance readability and comprehension. Suggestion: Don't use // comments for remarks that extend more than two lines, instead use /\* ... \*/ comments.
* All comments should be clear, grammatically correct, and use simple English phrases.
* Comments should be meaningful and relevant. They should assist the reader in comprehending the code by adding helpful annotations. The comment should not simply restate the code.   
  BAD:

           // Repeat 7 times

           for (int Day = 1; Day <= 7; Day++)

BETTER:

           //Process each day of the week

           for (Day=Days.Sunday; Day<=Days.Saturday; Days.Next(Day))

* A trailing comment after a close brace may be used to assist matching the closing brace with the corresponding open brace.
* The acronym "TBD" is commonly used to stand for "To Be Done" to indicate where required code has not yet been implemented.  A "TBD" comment may be used to acknowledge that something is missing and needs to be completed later.

# NAMING CONVENTIONS

Identifiers must have meaningful names, i.e., names that convey as much information as possible about what the object is or what the process does. E.g., totalPersons is better than count, and row is better than I.

Choose names for different identifiers that are "psychologically distant" in order to avoid confusion. E.g., maleTotal, femaleTotal is better than sum1, sum2.

Nouns are used for types, constants, variables. Verb phrases are used for methods. For example:   
     Variables: person carMake hoursWorked   
     Data Types: dayofWeek currency temperature   
     Methods: readName findAverage isWinner

Use constant identifiers in your program instead of "magic numbers."   
Example: if (Score > 80) // BAD! What does 80 mean?

final int PASSING\_GRADE = 80; // score needed to pass   
if (Score > PASSING\_GRADE) // OK

#### Capitalization

Class names must always start with a capital letter, e.g Hello, MyFirstClass

User defined variables are mixed case, with the first letter of every word in small, and NO underscores.   
     e.g. nameLength, highTemp, maxHeight

Method names are mixed case, with the first letter of every word in lower case, and NO underscores between words.   
     e.g. makeMove, showPrompt, isValidReply

Constant identifiers are upper case with words joined by an underscore ‘\_’

e.g. PASSING\_GRADE

## OTHER SUGGESTED STYLE CONVENTIONS

#### Boolean Expressions

Never use a boolean constant in a boolean expression.   
Example:  if (someFlag == TRUE) should be if (someFlag) .

#### Parentheses

Within expressions, no space after open parenthese, e.g.

(unit\_price \* quantity)

Within parameter lists, include space after open parenthesis, e.g.,

Find\_Average( first\_parm, second\_parm);

#### Braces {}

Matching open and closing braces must be **vertically aligned**. "Marker" comments may be used to assist matching the closing brace with the corresponding open brace. For example: // end loop

* **Always** use braces around the body of loops and decisions.

        if (Reply == 'y')

              PlayAgain();       // BAD

        if (Reply == 'y')

        {

              PlayAgain();      // OK

        }

# javadoc

Javadoc is a utility that comes in the JDK that automatically generates documentation from your source code. All you have to do is place comments in your code using certain descriptive tags, and then run the utility. It will create a nicely formatted HTML file that includes your comments. Importantly, the file contains hyperlinks to related classes and methods. Your comments must be in the following form (note *two* asterisks at start):

/\*\*

\* purpose

\*  
 \* @param name description   
 \* @param name description   
 \* @return description

\*/

From a command line, you invoke the javadoc utility with the command

javadoc MyProgram.java

A big advantage of javadoc is that it lets you keep the program documentation embedded in the code, rather than a separate external document that gets lost or goes out of date.   
 