1.0 Programming Style Guide

This document provides a standard so that code generated will have a consistent look and feel from person to person. This will help you read the code provided by your instructors and classmates; it will also help your instructors and classmates read your code. This style is not the only way a program can be written, and in other organizations, you may find that a different style guide will be used. However, for programs written while you are in Computer Systems Technology, you must follow these guidelines, unless otherwise directed by your course instructors (they will provide an exception sheet, as required).

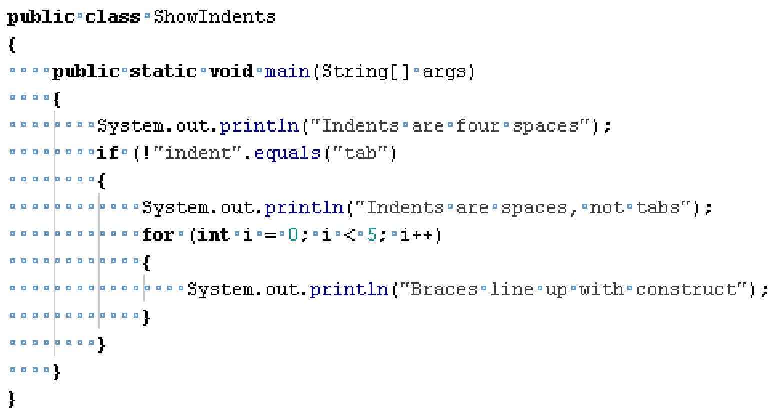
There may be some features discussed that you have not yet encountered in your programming experience. If you are not using these constructs, do not worry about following those rules! If necessary, your instructors will let you know what aspects of the guide are important to you.

1.1 Formatting:

1.1.1 Braces:

All braces must line up. Braces start in the same column as their construct. (See Figure 1.)

Figure 1



```
public class ShowIndents
{
    public static void main(String[] args)
    {
        System.out.println("Indents are four spaces");
        if (!"indent".equals("tab")
        {
            System.out.println("Indents are spaces, not tabs");
            for (int i = 0; i < 5; i++)
            {
                System.out.println("Braces line up with construct");
            }
        }
    }
}
```

1.1.2 Spacing:

All identifiers and binary operators are surrounded by white space. Parentheses for method calls and array index operators do not require extra space

```
int aValue=5; // no
int aValue = 5; // preferred

if(aValue<3) // no
if (aValue < 3) // preferred

int newSalary = employee.giveRaise(0.05); // extra spaces around ( ) not necessary

nextItem = arrayValues[i]; // extra spaces around [ ] not necessary

// sometimes spaces will help group items - use judiciously
if ( ( year % 400 == 0 || ( year % 4 == 0 && !(year % 100 == 0) ) )
```

1.1.3 Blank Lines:

Long chunks of code should be broken up with blank lines, where appropriate. Of course, long chunks of code should be avoided, where possible! As a general rule of thumb, if a method cannot fit onto one screen (40 lines at most), consider breaking the method into calls to smaller methods. (This may not be practical in all situations.)

Each method should be separated from the next with two blank lines.

1.1.4 Java File Organization:

Only have one class per file. The file should be organized as follows:

- package statement (if applicable)
- import statements (if applicable)
- non-javadoc class documentation (if applicable)
- javadoc class documentation
- Class definition, organized as follows:
    - final attributes
    - attributes
    - constructors
    - methods

1.1.5 Maximum Line Length:

Most of your code should be under 80 characters per line. ~~No line should exceed 120 characters.~~ If you find your code is getting indented too far to the right, consider breaking your code into more methods.

1.1.5.1 Wrapping Lines:

When an expression will not fit on a single line, try to break the line according to the following principles:

- break after a comma (e.g. in method headers, method calls)
- break ~~before~~ after an operator
- indent the following line by eight spaces after the current line
- prefer higher-level breaks to lower-level breaks (i.e. break outside a parenthesized expression rather than within the expression, if possible)

1.1.6 Parentheses:

Use parentheses to clarify order of precedence and simplify expressions. If you are unsure of what the precedence is, the use them to be sure!

1.2 Identifiers:

All identifiers (class names, variable names) should consist of the letters ‘a’ through ‘z’, ‘A’ through ‘Z’, and numbers ‘0’ through ‘9’. An underscore character (‘_’) may be used for constants. All other symbols and non-ASCII characters should be avoided.

All identifiers should reasonably indicate the purpose of the named item. The only exceptions are variables for simple loop counters, such as i, j, and k.

1.2.1 Package Names:

Package names (if used) should use lowercase only. Try to keep the length under eight characters, and avoid multi-word package names.

1.2.1.1 Class and Interface Names:

All class and interface names will use mixed case. The first letter of each word, including the first word, should be capitalized. All other letters will be lowercase. In the case where one of the words is an acronym, the acronym should be all uppercase.

```
MySpiffyClass
CSTStudent
ClassForCST
```

1.2.2 All Other Identifiers (Method and Variable Names):

All other identifiers, including attributes, local variables, methods, and parameters will use mixed case. The first letter of the identifier will be lowercase. The first letter of subsequent words will be uppercase. All other letters will be lowercase, except in the case of an embedded acronym. Spell named constants using all uppercase letters.

```
customer
studentIDNumber
cstInstructor
public static final int NUM_OF_STUDENTS = 24;
```

1.2.3 Basic Method Names:

Methods that set object state (mutators or “transformers”) should start with “set”. Methods that retrieve object state (accessors) should start with “get”. A method that retrieves a boolean value should start with “is”.

```
setCSTInstructor
getStudentIDNumber
isFriday
```

1.3 Comments:

All code requires comments! Every class, every method, and every non-private attribute should have an appropriate comment describing its purpose and use. Complex pieces of code should have proper documentation to detail design decisions and/or algorithms.

Java supports three types of comments:

// in-line comment: runs from the two slashes to the end of the line
/* block comment: runs from the slash-star to the star-slash and can span multiple lines */
/** JavaDoc comment: runs from the slash-star-star to the star-slash and can span multiple lines. The javadoc utility can read these comments and create html documentation pages */

C++ supports the in-line and block comment styles.

Visual Basic and Visual Basic for Applications support in-line comments using a single quote (').

Oracle’s SQL supports in-line comments using a double dash (--) and block comments using /* */.

1.3.1 Class/File Comments:

Each class/file must have a comment detailing the following (Java programs use JavaDoc style):

- a brief statement of the purpose of the class
- a detailed description of how the class is used
- name(s) and CST number(s) of the programmer(s) (JavaDoc using one or more @author tags)
- version number (JavaDoc using the @version tag)

Each class should also have a comment with the following details (Java programs use non-JavaDoc):

- a description of the requirements, design specification, and your solution strategy
- date submitted
- assignment number
- course name
- name of your instructor(s)
- file path and name

1.3.2 Instance Variable Comments:

Each instance variable should have a description of its purpose. For private variables, an in-line comment is sufficient. For Java programs with non-private variables, the comment should be in JavaDoc format.

1.3.3 Method Comments:

Each method must have a comment detailing the following (Java programs use JavaDoc style):

- a brief statement of the purpose of the class
- a description of each parameter (JavaDoc using @param tags)
- a description of the value returned (JavaDoc using the @return tag)
- a description of any exceptions thrown (JavaDoc using @exception tags)

Local variables should be named to describe their purpose. If further explanation is required, then include in-line comments.

You should also describe your algorithm, embedded with the code (indented to the same column as the code is). In-line comments are sufficient for this purpose.

1.4 General Java Coding Conventions:

- Do not compound increment and decrement operators, unless you are planning on entering an obfuscated code contest. Putting the step on an extra line makes your code that much more readable to others.
- Make all class attributes private. If others need to access the attribute, provide non-private methods to access and modify the values.