Java Style Guide

This style guide is to be used as a reference for all work being done related to the AlarmBuddy project. The attached guides were heavily inspired by style guidelines provided by Google under the CC-By 3.0 License, which encourages the sharing of these and the original source documents. The original style guides provided by Google can be found at https://google.github.io/styleguide/

Table of Contents

Introduction

This document is a set of style guidelines for the AlarmBuddy's Alarm Clock Project.

Guiding Principles

These guidelines are put in place in order to maximize:

1. Readability
2. Debuggability
3. Maintainability
4. Consistency

Motivation

The purpose of this document is to ensure code written is done in the same way to make it easier to understand, review, debug, and maintain. The guidelines will range from topics such as indentation to naming methods and variables.

Source Files

Copyright Notice

Files should contain the appropriate license or copyright information at the very top of the file.

```
1  /**
2   * Copyright (c) year, who. All rights reserved
3   */
```

Package Declaration

The package statement should never be line-wrapped, even if it exceeds the column limit.

Import Statements

The import statements should never be line-wrapped, even if they exceed the column limit.
Wild card imports should not be used.
All static imports should be listed in a single block.
All non-static imports should be listed in a single block.
If there are both static and non-static imports, the static imports should be listed first, with a single blank line separating the static and non-static imports.
The names should appear in ASCII order within each block.
Static import should not be used for static nested classes, rather they should be imported with normal imports.

Class Declaration

Every top-level class should be in its own source file.
Each class should have some logical order. So, new methods should be placed somewhere that makes sense, rather than simply at the end of the class. Also, overloaded methods should never be split apart.
For example, this:

```
13      void setAmount(int i) {
14          amount = i;
15      }
16
17      void setAmount(double d) {
18          amount = d;
19      }
20
21      void addThem(int x, int y) {
22          total = x + y;
23      }
```

would make more sense than this:

```
13    void setAmount(int i) {
14        amount = i;
15    }
16
17    void addThem(int x, int y) {
18        total = x + y;
19    }
20
21    void setAmount(double d) {
22        amount = d;
23    }
```

Special Characters

The line terminator sequence and ASCII horizontal space character are the only two whitespace characters that are allowed. All others should be written I escaped form and tabs should not be used. If a character has a special escape sequence, use the sequence rather than the corresponding octal or Unicode escaped characters.

Use: \b, \t, \n, \r, \f, \', \", \\	**Don't Use:** \047, \012, \u000a \u0027

When deciding whether to use the actual Unicode character or equivalent Unicode escape, choose the one that makes the most sense. If one is easier to read, use that one. Otherwise, include a comment explaining what character it is.

Formatting

Braces

When using if, else, for, do, and while statements, braces should be used (even if the body is empty or only one line).

Nonempty blocks should not have a line break before the opening brace. However, they should have a line break after the opening brace, before the closing brace, and after the closing brace if that brace terminates a statement or the body of a method, constructor, or named class.

Like this:

```
25    public void myMethod() {
26        if (thisConditionIsMet()) {
27            try {
28                somethingToTry();
29            } catch (ProblemException e) {
30                recover();
31            }
32        } else if (thisOtherConditionIsMet()) {
33            doThisInstead();
34        } else {
35            doThis();
36        }
37    }
```

An empty block may have a closing bracket directly following the open bracket as long as it is not part of a multi-block statement.
These are both allowed:

```
40      void toDo() {}
41
42      void toDoLater() {
43      }
```

But this is not allowed:

```
48      try {
49          tryThis();
50      } catch (Exception e) {}
51
```

Indentation

Every time a new block starts, the indentation should increase by one tab. When the block ends, the indentation should decrease by one tab and return to the previous indentation level. This applies to both the code and comments of each block.

One Statement Per Line

Every statement should be followed by a line break.

Column Limit

The column limit is 100 characters. Any line that is longer than 100 characters should be line-wrapped.
Exceptions:

1. Lines where it is impossible to obey the column limit
2. package and import statements
3. Command lines in the comments that need to be cut-and-pasted directly into a shell

Line-wrapping

When lines exceed the column limit, the code is often more difficult to read. In order to maintain readability, there are preferred times to line break.

1. The break should come before the symbol when the break occurs at a non-assignment operator.

The following symbols are included in this rule as well:	**Reference:**
The dot separator	.
The two colons of a method reference	::
An ampersand in a type bound	`<T extends Fruit & Runnable>`
A pipe in a catch block	`catch (FruitException | RunnableException e)`

2. The break should come after the symbol when the break occurs at an assignment operator.
3. Do not detach the open parenthesis from the method or constructor name.
4. Commas should stay attached to the item before it.
5. If the body of a lambda consists of a single unbraced expression, then you may break immediately after the arrow. Otherwise, lines should never be broken before or after the arrow.

Both are acceptable:

```
46      (x, y) -> {
47          System.out.println(x);
48          System.out.println(y);
49          return (x + y);
50      };
51
52      Function<String> name = p ->
53          p.getFirstName();
```

Whitespace

A single blank line should separate:

1. Consecutive members or initializers of a class
    a. Fields
    b. Constructors
    c. Methods
    d. Nested classes
    e. Static initializers
    f. Instance initializers
2. Anywhere else this document advises vertical whitespace

Exceptions:

- Blank lines between two consecutive fields are optional.
- Black lines between enum constants are covered at a later point in this document.

One singular blank line may be added anywhere else that it improves readability. However, the use of multiple consecutive blank lines is not encouraged.

A single space should be used:

1. To separate reserved words from the open parenthesis directly following it
    a. if, for, catch, etc.
2. To separate reserved words from the closing curly brace directly before it
    a. else, catch, etc.
3. Before open curly braces

    Exceptions:

    -
    ```
    @Something({x, y})
    String[][] hello = {{'hello'}};
    ```
    -
4. Before and after binary and ternary operators

    Includes:

    - The ampersand in a conjunctive type bound
      `<T extends Fruit & Runnable>`
    - The pipe for a catch block that handles multiple exceptions
      `catch (FruitException | RunnableException e)`
    - The colon in a “foreach” statement
      `for (int variable : collection)`
    - The arrow in a lambda expression
      `(x, y) -> {`

    Exceptions:

- The two colons of a method reference
  `variable::toString`
- The dot separator
  `variable.toString();`

5. After , : ; or the closing parenthesis of a cast
6. Before and after a double slash (end-of-line comment)
7. In the middle of the type and variable when declaring
   `List<String> myList;`
8. In the middle of a type annotation and [ ] or . . .

Space is optional inside the braces of an array initializer
Both are valid:

```
int[] numbersA = {1, 2};

int[] numbersB = { 1, 2 };
```

These rules primarily relate to interior space. They do not relate to space at the start of a line or at the end of a line.
There is no need to use horizontal alignment when commenting.

Redundant Parentheses

Only use redundant parentheses when it improves the readability. For example, using redundant grouping parentheses in longer expressions may be helpful. However, using parentheses around the item being returned in a return statement is unnecessary.

Enum Classes

Line breaks are optional after the comma following each enum constant.
Both are acceptable:

```
private enum Season { WINTER, SPRING, SUMMER, FALL }


private enum eightBallResponse {
    TRY_AGAIN {
        @Override public String toString() {
            return "try again";
        }
    },

    YA,
    NAH
}
```

Variable Declarations

Only one variable per declaration and one declaration per line.

Exception:

- Multiple variables can be declared in the header of a for loop

Variables should be declared right before they are used and initialized as close to the declaration as possible.

Arrays

Using block style is a valid way to initialize arrays.
All are allowed:

```
new int[] {
    0,
    1,
    2,
    3
}
```

```
new int[] {
    0, 1,
    2, 3
}
```

```
new int[] {
    0, 1, 2, 3
}
```

```
new int[]
    { 0, 1, 2, 3}
```

C-style array declarations are not allowed.

Allowed: `String[] args`	**Not Allowed:** `String args[]`

Switch Statements

Similar to other blocks, switch statements are also indented with tabs.
Following a switch label, there should be a line break and the indentation level should increase by one tab. Once the switch label ends, the indentation level should decrease by one tab, and return to the previous indentation level.

Example:

```
switch (user) {
    case 18:
    case 19:
        user18or19();
        // fall through
    case 20:
        user18or19or20();
        break;
    default:
        otherUser(user);
}
```

Every switch statement must have a default statement group, even if it is blank.

Exception:

- If the switch statement is dealing with enums and includes explicit cases that cover every possible value, the default case may be omitted.

Annotations

If there are multiple annotations being applied to a class it should look like this:

```
@Override
@Nullable
public String myAnnotation() {

}
```

Exceptions:

- Single parameterless annotation

```
@Override public int addThem() {

}
```

- Annotations that apply to a field

```
@Partial @Mock Dataloader theLoader;
```

Comments

The following are all valid ways to comment:

```
/*
* This is a
* comment.
*/

// This is a
// comment.


/* This is
*  a comment. */
```

Modifiers

Modifiers should appear in this order:

1. public
2. protected
3. private
4. abstract
5. default
6. static
7. final
8. transient
9. volatile
10. synchronized
11. native
12. strictfp

Numeric Literals

Integer literals that are the long type should always be followed by an L (not a lowercase l).

Allowed: `long x = 200000L;`	**Not Allowed:** `long y = 200000l;`

Naming

All identifiers should only use ASCII letters and digits (in certain situations underscores are allowed as well). Use descriptive names for important identifiers and brief names for nonimportant identifiers (such as local short-lived variables). Never use special prefixes or suffixes.

Package Names

These should be all lowercase. Use a . when multiple words are necessary.

Allowed: com.example.thename	**Not Allowed:** com.example.theName & com.example.the_name

Class Names

These should be written in upper camel case. Use nouns or noun phrases when naming classes. Interface names can also be adjectives or adjective phrases,

Method Names

These should be written in lower camel case. Use verbs or verb phrases when naming methods. When naming Junit test methods, underscores are allowed. Generally they look like this: <methodUnderTest>_<state>.

Constant Names

These should be written in all uppercase letters with each word separated by an underscore. Try to use nouns or noun phrases when naming constants.
This includes:

- primitives
- Strings
- immutable types
- immutable collections of immutable types

Non-Constant Field Names

These should be written in lower camel case. Use nouns or noun phrases when naming non-constant fields.

Parameter Names

These should be written in lower camel case. Refrain from using one character when naming parameters in public methods.

Local Variable Names

These should be written in lower camel case. Even if a local variable is final and immutable, it should be styled as a local variable rather than a constant.

Type Variable Names

These are either named a single capital letter (and sometimes a single numeral as well) or a name in the form used for classes followed by the capital letter T.

Camel Case

How to convert an English phrase into camel case:

1. Convert the phrase to plain ASCII and remove all apostrophes
2. Divide the phrase into words (split on spaces and remaining punctuation)
    a. If a word already has camel case conventions, split into parts (“YouTube” becomes “you tube”)
3. Lowercase everything
4. Uppercase the first character of every word
    a. If yielding to lower camel case, keep the first letter of the first word lowercase
5. Join all the words together

Examples:

Phrase:	**Lower Camel Case:**	**Upper Camel Case:**
“St. Thomas Student ID”	stThomasStudentId	StThomasStudentId
“iOS Support”	iosSupport	IosSupport
“Number Line 1”	numberLine1	NumberLine1

Programming Practices

@Override

Whenever it is legal, @Override should be used.

- Class method overriding a superclass method
- Class method implementing an interface method
- Interface method respecifying a superinterface method

Exception:

- When the parent method is @Deprecated, @Override may be omitted

Caught Exceptions

Most times, it is not allowed to ignore caught exceptions. However, when it is truly appropriate, the reasoning must be justified in a comment.

Exception:

- Caught exceptions may be ignored and no comment is needed when its name is or begins with expected

Static Members

When a reference to a static class member must be qualified, it is qualified with that specific class’s name rather than with a reference relating to that class’s type.

Finalizers

Do not override Object.finalize.

Javadoc

Formatting

If there are no block tags and the Javadoc block can fit on one line write it like this:

```
/* This is a single line. */
```

Otherwise write it in this format:

```
/*
* Multiple lines should be written
* like this.
*/
```

A singular blank line should appear between paragraphs and before block tags. Every paragraph, other than the very first paragraph, should have a <p> immediately before the first word.
Every block tag should have a description and appear in this order:

1. @param
2. @return
3. @throws
4. @deprecated

If a block tag needs multiple lines, the continuation lines should be indented at least four spaces after the associated @.

```
/*
* @return this is a very, very, very
*         long description of what
*         is being returned
*/
```

Summary Fragment

Every Javadoc block should have a summary fragment at the beginning. It should not be written in complete sentences, but it should be formatted similarly with capitalization and punctuation. It is generally comprised of noun or verb phrases.

```
/* Returns the student ID number. */
```

Where to Use

A Javadoc block should be used for every public class and public or protected member of the public class.

Exceptions:

- A method that is self-explanatory
- A method that overrides a supertype method

If it would be useful to define another class or member, Javadoc blocks are encouraged.