**War of Robotcraft**

**Coding Style**

Team: A3

Team Members:

|  |  |  |
| --- | --- | --- |
| **Name** | **NSID** | **Student ID** |
| Fu, Chen | chf354 | 11183491 |
| He, Jiahuan | jih889 | 11183346 |
| Wang, Shisong | shw940 | 11157916 |
| Xie, Ruida | rux793 | 11194258 |
| Yang, Chen | chy202 | 11183550 |

Date: Nov 3, 2016

1. Introduction

This document is used as the guideline for coding standard of War of RobotCraft. It is based on Google Java Style Guide, which references to [https://google.github.io/styleguide/javaguide.html - s2.2-file-encoding](https://google.github.io/styleguide/javaguide.html#s2.2-file-encoding).

1. Source file structure

A source file consists of, in order:

* Package statement
* Import statements
* Exactly one top-level class
  1. Package statement

The package statement is not line-wrapped. The column limit dose not apply to package statements.

* 1. Import statements
     1. No wildcard imports

Wildcard imports, static or otherwise, are not used.

* + 1. No line-wrapping

Import statements are not line-wrapped. The column limit does not apply to import statements.

* 1. Class declaration
     1. Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

* + 1. Class member ordering

The ordering is as following:

* Fields;
* Constructor;
* Setters and getters;
* Private methods block;
* Public methods block.

1. Formatting
   1. Braces
      1. Braces are used where optional

Braces must be used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.

* + 1. Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("[Egyptian brackets](http://www.codinghorror.com/blog/2012/07/new-programming-jargon.html)") for nonempty blocks and block-like constructs:

* No line break before the opening brace;
* Line break after the opening brace;
* Line break before the closing brace;
* Line break after the closing brace, only if that brace terminates a statement or terminates the body of a method, constructor, or named class. For example, there is no line break after the brace if it is followed by else or a comma.
  + 1. Empty blocks: may be concise

An empty block or block-like construct may be closed immediately after it is opened, with no characters or line break in between ({}), unless it is part of a multi-block statement (one that directly contains multiple blocks: if/else-if/else or try/catch/finally).

* 1. Block and line indentation: + standard tab (4 spaces)

Each time a new block or block-like construct is opened, the indent increases by standard tab (4 spaces). When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

* 1. One statement per line

Each statement is followed by a line break.

* 1. Column limit: 100

Java code has a column limit of 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped.

Exceptions:

* Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference);
* package and import statements (see Sections 2.1 Package statement and 2.2 Import statements);
* Command lines in a comment that may be cut-and-pasted into a shell.
  1. Line-wrapping

There is no comprehensive, deterministic formula showing exactly how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

* + 1. Where to break

The prime directive of line-wrapping is: prefer to break at a higher syntactic level. Also:

When a line is broken at a non-assignment operator the break comes before the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)

* This also applies to the following "operator-like" symbols: the dot separator (.), the two colons of a method reference (::), the ampersand in type bounds (<T extends Foo & Bar>), and the pipe in catch blocks (catch (FooException | BarException e)).
* When a line is broken at an assignment operator the break typically comes after the symbol, but either way is acceptable.
* This also applies to the "assignment-operator-like" colon in an enhanced for ("foreach") statement.
* A method or constructor name stays attached to the open parenthesis (() that follows it.
* A comma (,) stays attached to the token that precedes it.
  + 1. Indent continuation:

Following indentation style (3.2 Block and line indentation: + standard tab (4 spaces) ).

* 1. Whitespace
     1. Vertical Whitespace

A single blank line appears:

1. Between consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.

* Exception: A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create logical groupings of fields.
* Exception: Blank lines between enum constants are covered.

1. Between statements, as needed to organize the code into logical subsections.
2. As required by other sections of this document (such as Section 3, Source file structure, and Section 3.3, Import statements).
   * 1. Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places only.

* Separating any reserved word, such as if, for or catch, from an open parenthesis (() that follows it on that line
* Separating any reserved word, such as else or catch, from a closing curly brace ()) that precedes it on that line
* Before any open curly brace ({), with two exceptions:

@SomeAnnotation({a, b}) (no space is used)

String[][] x = {{"foo"}}; (no space is required between {{, by item 8 below}

* On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:

the ampersand in a conjunctive type bound: <T extends Foo & Bar>

the pipe for a catch block that handles multiple exceptions: catch (FooException | BarException e)

the colon (:) in an enhanced for ("foreach") statement

the arrow in a lambda expression: (String str) -> str.length()

but not

the two colons (::) of a method reference, which is written like Object::toString

the dot separator (.), which is written like object.toString()

* After ,:; or the closing parenthesis ()) of a cast
* On both sides of the double slash (//) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
* Between the type and variable of a declaration: List<String> list
  1. Specific constructs
     1. Variable declarations
        1. One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as int a, b; are not used.

* + - 1. Declared when needed

Local variables are not habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

* + 1. Arrays
       1. Array initializers:

All in one single line under line limit.

* + 1. Switch statements
       1. Indentation

Following indentation style (3.2 Block and line indentation: + standard tab (4 spaces)).

* + - 1. Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a break, continue, return or thrown exception), or is marked with a comment to indicate that execution will or might continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically // fall through). This special comment is not required in the last statement group of the switch block.

* + - 1. The default case is present

Each switch statement includes a default statement group, even if it contains no code.

* + 1. Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, Line-wrapping), so the indentation level is not increased. Example:

* + 1. Comments

Block comments are indented at the same level as the surrounding code. They may be in /\* ... \*/ style or // ... style. For multi-line /\* ... \*/ comments, subsequent lines must start with \* aligned with the \* on the previous line.

* + 1. Modifier

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

public protected private abstract default static final transient volatile synchronized native strictfp

* + 1. Numeric Literals

long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1). For example, 3000000000L rather than 3000000000l.

1. Naming
   1. Rules common to all identifiers

Take camel case naming standard.

* 1. Rules by identifier type
     1. Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, com.example.deepspace, not com.example.deepSpace or com.example.deep\_space.

* + 1. Class names
* Class names are written in UpperCamelCase.
* Class names are typically nouns or noun phrases. For example, Character or ImmutableList. Interface names may also be nouns or noun phrases (for example, List), but may sometimes be adjectives or adjective phrases instead (for example, Readable).
* There are no specific rules or even well-established conventions for naming annotation types.
* Test classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest.
  + 1. Method names
* Method names are written in lowerCamelCase.
* Method names are typically verbs or verb phrases. For example, sendMessage or stop.
* Underscores may appear in JUnit test method names to separate logical components of the name. One typical pattern is test<MethodUnderTest>\_<state>, for example testPop\_emptyStack. There is no One Correct Way to name test methods.
  + 1. Constant names
* Constant names use CONSTANT\_CASE: all uppercase letters, with words separated by underscores. But what is a constant, exactly?
* Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really feels like a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely intending to never mutate the object is generally not enough.
  + 1. Non-constant field names
* Non-constant field names (static or otherwise) are written in lowerCamelCase.
* These names are typically nouns or noun phrases. For example, computedValues or index.
  + 1. Parameter names
* Parameter names are written in lowerCamelCase.
* One-character parameter names in public methods should be avoided.
  + 1. Local variable names
* Local variable names are written in lowerCamelCase.
* Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.
  + 1. Type variable names
* Each type variable is named in one of two styles:
* A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
* A name in the form used for classes (see Section 5.2.2, Class names), followed by the capital letter T (examples: RequestT, FooBarT).

1. Javadoc
   1. Formatting
      1. General form

The basic formatting of Javadoc blocks is as seen in this example:

/\*\*

\* Multiple lines of Javadoc text are written here,

\* wrapped normally...

\*/

public int method(String p1) { ... }

... or in this single-line example:

/\*\* An especially short bit of Javadoc. \*/

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

* + 1. Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (\*)—appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has <p> immediately before the first word, with no space after.

* + 1. At-clauses

Any of the standard "at-clauses" that are used appear in the order @param, @return, @throws, @deprecated, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the @.

* + 1. The summary fragment

The Javadoc for each class and member begins with a brief summary fragment. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does not begin with A {@code Foo} is a..., or This method returns..., nor does it form a complete imperative sentence like Save the record.. However, the fragment is capitalized and punctuated as if it were a complete sentence.

Tip: A common mistake is to write simple Javadoc in the form /\*\* @return the customer ID \*/. This is incorrect, and should be changed to /\*\* Returns the customer ID. \*/.

* + 1. Where Javadoc is used

At the minimum, Javadoc is present for every public class, and every public or protected member of such a class, with a few exceptions noted below.

Additional Javadoc content may also be present, as explained in Section 7.3.4, Non-required Javadoc.

* + 1. Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like getFoo, in cases where there really and truly is nothing else worthwhile to say but "Returns the foo".

Important: it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named getCanonicalName, don't omit its documentation (with the rationale that it would say only /\*\* Returns the canonical name. \*/) if a typical reader may have no idea what the term "canonical name" means!

* + 1. Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.

* + 1. Non-required Javadoc

Other classes and members have Javadoc as needed or desired.

Whenever an implementation comment would be used to define the overall purpose or behavior of a class or member, that comment is written as Javadoc instead (using /\*\*).

Non-required Javadoc is not strictly required to follow the formatting rules of Sections 7.1.2, 7.1.3, and 7.2, though it is of course recommended.