Coding Standard Document

This document contains the coding conventions created for Dungeons & Dragons project, based on the Java language coding standards presented in the *Java Language Specification*, from Sun Microsystems, Inc.

File Names

As in the Java language coding standards, source files have the .java extension and complied bytecode files have the .class extension.

Java Source Files

Each Java source file contains a single class or interface.

Java source files have the following ordering:

- Beginning comments
- Package and Import statements
- Class Header and Declaration
- Method Headers and Declarations

Beginning Comments

Class names such asFilename.java have Filename matching class name.

Package and Import Statements

The first non-comment line of most Java source files is a `package` statement. After that, `import` statements can follow. For example:

```
packagedungeons_and_dragons.controller;
importjava.awt.Color;
```

Class Headers and Declaration

All source files begins with a **JavaDoc** documentation which specifies author who has worked for the class and related information about the class file like functionality of the whole class.

```
/**
 *  Class name:
 *  Functionality:
 */
 public class ClassName
```

Method Headers and Declarations

Every method included in a class should contain a JavaDoccomment that lists the functionality,@param, @throws, @return which it supports, if any.

```
/**
 * Functionality:
 * @param:
 * @throws:
 * @return:
 */
```

Indentation

One Tab should be used as the unit of indentation.

Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- if the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Comments

Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

- **Block Comments**

  Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

- **Single-line comments**

  Short comments can appear on a single line indented to the level of the code that follows. A single-line comment should be preceded by a blank line.

Declarations

- **Number Per Line**

One declaration per line which is useful for commenting. Variable names should be sorted by type and in alphabetical order. Example:

```
int     level;      // indentation level
int     size;       // size of table
String  stringA;
String  stringB;
```

- **Initialization**

  Local variables are initialized where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

- **Placement**

  Declarations are only put at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".)

- **Class and Interface Declarations**

  When coding Java classes and interfaces, the following formatting rules should be followed:

  - No space between a method name and the parenthesis "(" starting its parameter list
  - Open brace "{" appears at the end of the same line as the declaration statement
  - Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
        int ivar1;
        int ivar2;

        Sample(inti, int j ) {
          ivar1 = i;
          ivar2 = j;
                }

                intemptyMethod() {}
                ...
        }
```

  - Methods are separated by a blank line.

Statements

- **Simple Statements**

  Each line should contain at most one statement. Example:

```
argv++;                 // Correct
argc--;                 // Correct
argv++; argc--;         // AVOID!
```

- **Compound Statements**

  Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

  - The enclosed statement should be indented one more level than the compound statement.
  - The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

➢ **return Statements**

A `return` statement with a value should not use parentheses. Example:

```
returnreturnValue;
```

➢ **if, if-else, if else-if else Statements**

The `if-else` class of statements should have the following form:

```
if (condition ) {
        statements;
}

if ( condition ) {
        statements;
} else {
  statements;
}

if (condition ) {
        statements;
} else if ( condition) {
      statements;
} else {
     statements;
}
```

➢ **for Statements**

A `for` statement should have the following form:

```
for ( initialization; condition; update ) {
    statements;
}
```

➢ **while Statements**

A `while` statement should have the following form:

```
while ( condition ) {
```

statements;
}

- **switch Statements**

A `switch` statement should have the following form:

```
switch ( condition ) {
case ABC:
        statements;
   /* falls through */

case DEF:
   statements;
break;

case XYZ:
   statements;
break;

default:
   statements;
break;
}
```

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

- **try-catch Statements**

A `try-catch` statement should have the following format:

```
try {
  statements;
} catch ( ExceptionClass e ) {
     statements;
}
```

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Between logical sections inside a method to improve readability

Blank Spaces

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = ( a + b ) / ( c * d );

while ( d++ = s++ ) {
  n++;
}
printSize( "size is " + foo + "\n" );
```

Naming Conventions

- **Class Names**

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Class names should be simple and descriptive. Use whole words-avoid acronyms and abbreviations. Examples:

classMapValidator; class ItemController;

- **Method Names**

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized. Examples:

save();

Getters and Setters that are auto-generated can consist of "_" with the attributes that are kept private. Example:

getMap_entry_door();

setMap_wall();

- **Variable Names**

Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters. Example:

intmap_id;

- **Constant Names**

  The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.) Example:

```
public static final String INTELLIGENCE = "Intelligence";
```