UNIVERSITÉ Concordia UNIVERSITY

SOEN 6441

Advanced Programming Practices

Coding Standards Document

Team 20

Konstantin Chemodanov…. 40049285
Farzaneh Jamshidi…..…...…40075234
Saman Safaei………………. 40119399
Ali Molla Mohammadi …….40042597
Negar Ashouri………………40071530

March 2019

Department of Computer Science and Software Engineering

Coding Standard Document

This document describes the coding conventions of "Risk Game" project, based on the Oracle Java language coding standards presented in the **Java Language Specification** and **Java Code Conventions**, from Sun Microsystems, Inc.

File Names:
Based on Java coding standards, source files have the *.java* extension.
Java Source Files:
Each Java source file contains a single class or interface.
Java source files have the following ordering:
1. Package and Import statements
2. Class Header and Declaration
3. Method Headers and Declarations

1. 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:

```java
package risk.game.grp.twenty.model;

import java.util.ArrayList;
import java.util.List;
```

2. Class Headers and Declaration
Source files begins with a **JavaDoc** documentation which specifies the authors who have worked for the class and related information about the class file like functionality of the whole class.

```java
/**
 * This class is the Model of <em>Player</em> Object in Game.
 *
 * @author Team 20
 */
public class Player {
```

3. Method Headers and Declarations:
Methods included in a class usually contain a Java Document that explains the functionality of that method and the parameters, what method returns and throws.

```java
/**
 * this method handle the exchangeCard request from UI and return the JSON response back
 *
 * @param playerNumber Player Number equivalent of the <em>Player Turn</em> in the game
 * @return <em>HttpStatus.OK</em> and list of Players in case of successful event,
 * <em>HttpStatus.BAD_REQUEST</em> if any rules of game being violated, or any exception happens
 * during the process with error message that guide the user and handled in UI
 */

@RequestMapping("/exchangeCard")
public ResponseEntity exchangeCard(@RequestParam String playerNumber) {
```

Indentation:
4 spaces should be used as the unit of indentation according to the standards.
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. 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(int i, 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:

```
arga++; // Correct
argb--; // Correct
arga++; argb--; // 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.
- Braces are used around all statements, even singletons, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

- **Return Statements**

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

```
Return Value;
```

- **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;
case DEF:
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;
}
```

White Space:

- **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. Example:

```
class ReinforcementPhaseEndpoint;
```

- **Method Names**

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

```
loadMap ();
getArmies();
```

- **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 tempX, tempY, tempZ for integers; c, d, and e for characters. Example:

```
int countryId;
```

- **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.)

- **Object Names**

The name of objects of class should be all in lowercase .
Example:

```
PlayerStatus playerStatus;
```

References

Oracle.com. (1997). *java code convention*. [online] Available at: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Docs.oracle.com. (n.d.). *Java SE Specifications*. [online] Available at: https://docs.oracle.com/javase/specs/