**Code Conventions for the Game Risk**

## 1-Introduction

## Coding Conventions are documented as a set of rules that use how to code are written. They are important to programmers because of some reasons:

* 80% of the lifetime cost of a piece of software goes to maintenance.
* Hardly any software is maintained for its whole life by the original author.
* Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
* If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

## 2- File Names

## This section lists commonly used file suffixes and names.

## 2.1 File Suffixes

## JavaSoft uses the following file suffixes:

|  |  |
| --- | --- |
| File Type | Suffix |
| Java source | .java |
| Java bytecode | .class |

## 2.2 Common File Names

## Frequently used file names include:

|  |  |
| --- | --- |
| File Name | Use |
| File project name | GameRisk |
| README | The preferred name for the file that summarises the contents of a particular directory. |

## 3 - File Organisation

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

## Files longer than 2000 lines are cumbersome and should be avoided.

## Java source files have the following ordering:

## 3.1 Beginning Comments

All source files should begin with a comment that lists the programmer(s), the date, a

copyright notice, and also a brief description of the purpose of the program. For example:

*/\*\*  
 \* This class represents the base class  
 \* for continent and country class  
 \* it maintains id and name properties  
 \** ***@author*** *Kourosh Aziz-Nejad  
 \** ***@version*** *1.0.0.0  
 \*/*

## 3.2 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:

**package** model;**import** java.util.ArrayList;  
**import** java.util.Iterator;  
**import** java.util.List;  
**import** java.util.Random;

## 3.3 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear.

|  |  |
| --- | --- |
| **Part of Class/Interface**  **Declaration** | **Notes** |
| Class/interface documentation  comment (/\*\*...\*/) | All of class in this project have a documentation comment . |
| Class/interface implementation  comment (/\*...\*/), if necessary | This comment should contain any class-wide or interface-wide information that wasn’t appropriate for the class/interface documentation comment. |
| Class (static) variables | First the public class variables, then the protected,  and then the private |
| Instance variables | First public, then protected, and then private. |
| Constructors | Public. On the name of the class. |
| Methods | These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier. |

## 4 - Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces.

## 4.1 Line Length

Avoid lines longer than 80 characters, since they’re not handled well by many terminals and tools.

## 4.2 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.

## 5 - Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /\*...\*/, and //. Documentation comments (known as “doc comments”) are Java-only, and are delimited by /\*\*...\*/. Doc comments can be extracted to HTML files using the Javadoc tool.

## 5.1 Implementation Comment Formats

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

## 5.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should 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.

A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk “\*” at the beginning of each line except the first.

/*\*  
 \* This class represents the map  
 \* it maintains the countries and continents   
 \* as a collection of lands  
 \* it has different methods to change the state  
 \*/*

Block comments can start with /\*-, which is recognized by indent(1) as the beginning of a block comment.

## 5.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here’s an example of a single-line comment in Java code

**public int** compareTo(Object o) {  
 *// TODO Auto-generated method stub* Edge edge = (Edge) o;  
 **if** ((**this**.GetCountryId1() == edge.GetCountryId1() && **this**.GetCountryId2() == edge.GetCountryId2())  
 || (**this**.GetCountryId1() == edge.GetCountryId2() && **this**.GetCountryId2() == edge.GetCountryId1())) {  
 **return** 0;  
 }  
 **else return** -1;  
}

## 5.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting. Avoid the assembly language style of commenting every line of executable code with a trailing comment.

Here’s an example of a trailing comment in Java code.

**if** (**this**.instance == **null**) /\* when instance is not equal null run this block of if \*/   
 {  
 players = **new** ArrayList<Player>();  
 map = **new** Map(**"map"**);  
 turnOrganizer = **new** TurnOrganizer();  
 }

## 5.2 Documentation Comments

A JavaDoc comment begins with the /\*\* marker and ends with the \*/ marker. All the lines in the middle start with an asterisk lined up under the first asterisk in the first line.

*/\*\*  
 \* This class represents a connection between two countries  
 \* it maintains id of two countries which are neighbor  
 \* it has different methods to change and set the state of the object  
 \** ***@author*** *Kourosh Aziz-Nejad  
 \** ***@version*** *1.0.0.0*

*\*/*

**5.2.1 Notes**

## JavaDoc does not provide a format for commenting elements within methods, i.e. the local variables and the computing going on inside the methods.  But you still can use the regular comments marks // or /\*..\*/, to comment this part of your program.

## 5.2.2 JavaDoc Tags

## There are a number of special tags we can embed with the JavaDoc comments. These tags start with the “at” symbol @.

## @author

## Used to create an author entry. You can have multiple @author tags. This tag is meaningful only for the class/interface JavaDoc comment.

## @version

## Used to create a version entry. A JavaDoc comment may contain at most one @version tag.

## @see

## Used to add a hyperlinked "See Also" entry to the class.

## @param

## Used to add a parameter description for a method. This tag contains two parts: the first is the name of the parameter and the second is the description. The description can be more than one line.

## @return

## Used to add a return type description for a method. This tag is meaningful only if the method’s return is non-void.

## @throws

## Used to describe an exception that may be thrown from this method. Note that if you have a throws clause, Javadoc will already automatically document the exceptions listed in the throws clause.

## {@inheritDoc}

## Used to copy the description from an overridden method.

## {@link *reference*}

## Used to link to another documented symbol, or to an URL external to the documentation.

## 6 - Declarations

## 6.1 Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

## int size; // size of table

## 6.2 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{” and “}”.) Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

**public boolean** DoesExistEdge(Edge prm\_edge)   
 {  
 **for**(Edge e : **this**.edges)  
 {  
 **if**(e.compareTo(prm\_edge)==0)  
 {  
 **return true**;  
 }  
   
 }  
 **return false**;  
}

6.3 Initialization

Try to initialize local variables 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.

## 6.4 Class and Interface Declarations

• 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

“{“

**public class** GameBoard {  
 **public** List<Player> players;  
 **public** Map map;  
 **public static** GameBoard instance;  
 **public** List<Integer> roundRobin;  
 **public** TurnOrganizer turnOrganizer;  
  
 **public** GameBoard() {

**if** (**this**.instance == **null**) {  
 players = **new** ArrayList<Player>();  
 map = **new** Map(**"map"**);  
 turnOrganizer = **new** TurnOrganizer();  
 }

• Methods are separated by a blank line

## 7 - Statements

## 7.1 Simple Statements

## Do not use the comma operator to group multiple statements unless it is for an obvious reason.

## Remark:

## There are two approaches:

## Maximize visibility of the different blocks by having curly braces alone on their line of code.

## Minimize code length by appending the open curly brace to the statement that precedes it.

## In our project, It is used the maximize visibility of different blocks

## Example:

**if**(c.GetId()==prm\_countryId)

{  
 **return true**;   
 }

## 7.2 if, if-else, if-else-if-else Statements

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

**if** (!DoesExistCountry(prm\_name))

{  
 **this**.lands.add(FactoryLand.GetLand(**"Country"**, prm\_name, prm\_continentId, prm\_x, prm\_y,-1));  
 **return "successful"**;  
}

**Else**

{  
 **return "duplicate"**;  
}

## 7.3for Statements

## A for statement should have the following form:

**for**(Continent c : **this**.GetContinents())

{  
 **if**(c.GetName().equals(prm\_name))

{  
 **return true**;   
 }  
}

**7.4 while Statements**

## A while statement should have the following form:

**while**(it.hasNext())

{  
 Object o = it.next();  
 **if**(((Edge)o).DoesContainCountry(prm\_countryId))

{  
 it.remove();  
 }

}

**7.6 switch Statements**

## A switch statement should have the following form:

**Switch**(type)  
{  
**case "Country"**:  
 **return new** Country(prm\_name, prm\_continentId, prm\_x, prm\_y);  
**case "Continent"**:  
 **return new** Continent(prm\_name, prm\_control);  
 **default**:  
 **return null**;  
}

## Every time a case falls through (doesn’t include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment.

## 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.

**7.7 try-catch Statements**

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

**try**{  
 result=MapParser.MapValidator(input);  
}  
**catch**(Exception ex)  
{  
 **throw** ex;  
}

## 8 - White Space

**8.1 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

## • Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment

## Between logical sections inside a method to improve readability

**8.2 Blank Spaces**

## Blank spaces should be used in the following circumstances:

## • A keyword followed by a parenthesis should be separated by a space. Example:

**while** (map.GetCountriesNotAssigned().size() > 0)  
{  
 **for** (Player p : players)  
 {  
 **if** (map.GetCountriesNotAssigned().size() > 0)  
 {  
 map.GetNotAssignedCountryRandom().SetPlayerId(p.GetId());  
 } **else  
 break**;  
 }  
}

## Note that a blank space should not be used between a method name and its opening

## parenthesis.

## The expressions in a for statement should be separated by blank spaces. Example:

**for**(**int** i=1; i<7; i++)  
{  
 players.add(**new** Player(i,**"player"**+i));  
}

## 9 - Naming Conventions

## Naming conventions make programs more understandable by making them easier to read.

## They can also give information about the function of the identifier

|  |  |
| --- | --- |
| Identifier Type | Rules for Naming |
| Filename | Use lower case characters only |
| Classes | Start with Project name |
| Methods | Start with a lower case letter and upper case letters to separate wordsRemarks: In our project the methods start with upper letter following Camel Script |
| Attributes | Use a lower case letter and use underscore between word |
| Variables | Local variables are written entirely lower case without underscore |
| Constants | Use upper case letter with underscore between words |