Project Coding Standard

Team 36

1. Phan Nguyen

2. Bin Xue

3. Xiyun Zhang

4. Yongxuan Zhang

5. Jiahui Wang

Table of Content

Background

The intention of this guide is to provide a set of conventions that encourage good code.

In general, much of our style and conventions mirror the Google's Java Style Guide, and slides from Dr. Joey Paquet .

Coding style

1. Formatting

1.1 Use line breaks wisely

There are generally two reasons to insert a line break:

1. Our statement exceeds the column limit.
2. We want to logically separate different thought and segment of code.

1.2 Indent style

We use the "one true brace style". Indent size is 4 columns.

Java

```java
// For example.
if (this.player.ifForceExchange()){
    //change card
}else{
    //do not change
}
```

We do not break up a statement unnecessarily.

Java

```java
final String value = otherValue;
```

Method declaration continuations.

Java

```java
// Preferred for easy scanning and extra column space.
//Using camel case
public String getName() {return name;}
```

1.3 Chained method calls

Java

```java
//   - The proper location for a new method call is unambiguous.
x2=e.getX();
y2=e.getY();
```

1.4 100 column limit

You should follow the convention set by the body of code you are working with. We tend to use 100 columns for a balance between fewer continuation lines but still easily fitting two editor tabs side-by-side on a reasonably-high resolution display.

1.5 No trailing whitespace

Trailing whitespace characters, while logically benign, add nothing to the program.

Java

```
private String matrixDisplayMode = "prefered"; //prefered, same
```

2.Field, class, method declarations and Variable naming

2.1 Extremely short variable names should be reserved for instances like loop indices.

Java

```
class Player {
  private int armies;
  private String name;
}
```

2.2 Include units in variable names

Java

```
//   - Unit is built in to the type.
//   - The field is easily adaptable between units, readability is high.
public ArrayList<Country> countries;
```

Also avoid embedding scope information in a variable. Hierarchy-based naming suggests that a class is too complex and should be broken apart.

Java

```
// Bad.
String _value;
String mValue;

// Good.
String value;
```

3. Space pad operators and equals.

Java

```
int foo = a + b + 1;
```

It's even good to be *really* obvious.

Java

```
if ((values != null) && (10 > values.size())) {
  ...
}
```

4. Absence of "commented out" codes

Delete codes between "/*" and "*/" to make codes clear.

Java

```
/*
public Country findCountry(String countryName) {
    for (ArrayList<Country> loopList : countries.values()) {
        for (Country loopCountry:loopList){
            if (loopCountry.countryName.equals(countryName)){
                return loopCountry;
            }
        }
    }
    return null;
}
*/
```

5. Javadoc

The more visible a piece of code is (and by extension - the farther away consumers might be), the more documentation is needed.

Java

```
// Bad.
/**
 * This is a class that implements a cache.  It does caching for you.
 */
class Cache {
  ...
}

// Good.
/**
 * A volatile storage for objects based on a key, which may be invalidated and discarded.
 */
class Cache {
  ...
}
```

5.1 Documenting a method

A method doc should tell what the method *does*. Depending on the argument types, it may also be important to document input format.

Java

```java
/**
 * Function to find the Country according to the Country's name
 * @param countryName Country's name
 * @return Country that found
 */
public Country findCountry(String countryName) {
    for (ArrayList<Country> loopList : countries.values()) {
        for (Country loopCountry:loopList){
            if (loopCountry.countryName.equals(countryName)){
                return loopCountry;
            }
        }
    }
    return null;
}
```

When a methid is too complicated to understand, it is better to add descriptions about how the method processes.

Java

```java
/**
 * Function to find the Country according to the Country's name
 *
 * <p> 1. loop the country arraylist (nested loop) <p> <br>
 * <p> 2. compare each country's name with targeted country's name <p><br>
 * <p> 3. return country if found or null if not <p> <br>
 *
 * @param countryName Country's name
 * @return Country that found
 */
public Country findCountry(String countryName) {
    for (ArrayList<Country> loopList : countries.values()) {
        for (Country loopCountry:loopList){
            if (loopCountry.countryName.equals(countryName)){
                return loopCountry;
            }
        }
    }
    return null;
}
```

5.2 Use javadoc features - No author tags

Code can change hands numerous times in its lifetime, and quite often the original author of a source file is irrelevant after several iterations. We find it's better to trust commit history and `OWNERS` files to determine ownership of a body of code.

6. Imports

6.1 Import ordering

Imports are grouped by top-level package, with blank lines separating groups. Static imports are grouped in the same way, in a section below traditional imports.

Java

```java
import java.*
import javax.*

import scala.*

import com.*

import net.*

import org.*

import com.twitter.*

import static *
```

6.2 No wildcard imports

Wildcard imports make the source of an imported class less clear. They also tend to hide a high class fan-out.
See also texas imports

Java

```java
// Bad.
import java.awt.event.ActionEvent;
import java.awt.event.*;

// Good.
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

7. Use interfaces

Interfaces decouple functionality from implementation, allowing you to use multiple implementations without changing consumers. Interfaces are a great way to isolate packages - provide a set of interfaces, and keep your implementations package private.

Many small interfaces can seem heavyweight, since you end up with a large number of source files. Consider the pattern below as an alternative.

Java

```java
interface FileFetcher {
  File getFile(String name);

  // All the benefits of an interface, with little source management overhead.
  // This is particularly useful when you only expect one implementation of an interface.
  static class HdfsFileFetcher implements FileFetcher {
    @Override File getFile(String name) {
      ...
    }
  }
}
```

7.1 Leverage or extend existing interfaces

Sometimes an existing interface allows your class to easily 'plug in' to other related classes.

Java

```java
class Blobs implements Iterable<byte[]> {
  @Override
  Iterator<byte[]> iterator() {
    ...
  }
}
```

Warning - don't bend the definition of an existing interface to make this work. If the interface doesn't conceptually apply cleanly, it's best to avoid this.

Writing testable code

Writing unit tests doesn't have to be hard. You can make it easy for yourself if you keep testability in mind while designing your classes and interfaces. Tests should be clear and relevant

Java

```java
/**
*
* Method: loadMapFile(String mapFileName)
*
*/
@Test
public void testLoadMapFile() throws Exception {
    assertEquals(true,map.loadMapFile("1.map"));
    map.clear();
    assertEquals(false,map.loadMapFile("2.map"));
    map.clear();
    assertEquals(false,map.loadMapFile("3.map"));
    map.clear();
    assertEquals(false,map.loadMapFile("4.map"));
    map.clear();
    assertEquals(false,map.loadMapFile("5.map"));
    map.clear();
}
```