SystemScape

Coding Standards

Introduction

All members present in this group must abide by the coding standards for all source codes in Java™ Programming Language as stated in this document. These coding standards are present to provide consistency in the quality of the group's source code.

Reminder:

The recommendations are all in the following format:

Recommendation

i.e.,

```
Examples if available
```

Table of Contents

1 *Naming Conventions*

1.1 General

Package names should always be written in lowercase.
i.e.,

```
designchallenge, view.calculator.event.single
```

Type names should always be nouns which are written in camel case beginning with an uppercase letter.
i.e.,

```
Animon, AnimonTrainer
```

Naming variables should always be in camel case beginning with a lowercase letter.
i.e.,

```
life, elementType
```

Constant names/ Final Variables should always be written in uppercase. Underscores must be used to separate words.
i.e.,

```
GEN_AVERAGE, THIRD_ITERATION
```

Method names must use verbs and should be written in camel case beginning with lower case.
i.e.,

```
updateCalendar(), getTotalScore()
```

Abbreviations and acronyms must not be written in uppercase when used as name.
i.e.,

```
openHtmlFile(); // NOT: openHTMLFile();
collectVcdPlayer(); // NOT: collectVCDPlayer();
```

Generic variable names and type names must be the same.
i.e.,

```
void catchAnimon(Animon animon) // NOT: void catchAnimon(Animon monmon)
                                  // NOT: void catchAnimon(Animon am)
void setName(String string) // NOT: void setName(String s)
                              // NOT: void setName(String temp)
```

All names should be in English.

Scratch variables or temporary storage variables should be named as *temp + their data type*.
i.e.,

```
int tempInt = 0;
String tempString = "temporary";
```

Object name is implicit and must be not be used in a method name.
i.e.,

```
animon.getType(); // NOT: animon.getAnimonType();
trainer.getName(); // NOT: trainer.getTrainerName();
```

1.2 Specific

Attributes should be accessed directly only through the use of the terms *get/set*
i.e.,

```
movie.getTitle();
movie.setTitle(title);
```

Complement names must be used in naming methods/variables if one entity must co-exist with its opposite.

i.e.,

get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume, etc.

```
int max;
int min;  //Not: int smallest; vice-versa

public void getFirst(){
   :
}
public void getLast(){  //Not public void getEnd(){
   :                       //
}                         //       } vice-versa
```

For boolean variables and methods, the prefix *is* must be used

i.e.,

```
isBooked, isAlive, isIdle, isAvailable
```

The JFC (Java Swing) element type should be appended to its variable name.

i.e.,

```
openButton, calendarTable, rightScrollPane, nameTextField
```

Collection of objects represented by names must be in plural form

i.e.,

```
ArrayList<Animon> animons;
int[] prices;
HashMap<String, String> cinemas;
```

The *suffix* "No" should be used for variables representing an entity number.
i.e.,

```
seatNo, screeningNo, packetNo
```

Use i, j, k etc. for iterator variables.
i.e.,

```
for (Iterator i = sales.iterator(); i.hasNext(); ) {
    :
}

for (int j = 0; j < nAnimons; j++) {
    :
}
```

Abbreviations should not be used in names.
i.e.,

```
JLabel nameLabel;  //NOT: JLabel nameLbl;
printScreen();  //NOT: prtScr();
```

Negated boolean variable names should not be used
i.e.,

```
boolean isAttending; // NOT: isNotAttending
```

***Exception* should be appended to Exception class names.**
i.e.,

```
class CannotIntoException extends Exception {

}
```

Factory classes must return its instances through the method named *"new[ClassName]"*

i.e.,

```
class BurgerFactory {

   public Burger newBurger(...) {

   }
}
```

---

2 *Files*

The case-sensitive name of the top-level class plus the .java extension is to be used as the filename of all source codes.

i.e.,

```
class Pokemon {
   public Pokemon() {

   }
}

*should be named as Pokemon.java
```

Classes should be stored in separate files wherein the filename matches the class name and declared as public.

Page break must be avoided.

Follow the guidelines in splitting lines.

i.e.,

```
totalPokemons = a + b + c +
                      d + e;

harvest(param1, param2,
          param3);

String name = "First name" + "Middle name" +
                    "Last Name”);
```

Guidelines in splitting lines:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

---

3 *Statements*

3.1 Package and Import Statements

The file’s first statement should be the package statement; every file must be included in a specific package.

Unless the package is made by SystemScape, explicitly list *all* imported classes.
i.e.,

```
import systemscape.*;

import javax.swing.JRadioButton;
import java.util.ArrayList;
```

3.2 Types

Array specifiers must be right next to the type not the variable.
i.e.,

```
int[] integers = new int[5];   // NOT: int integers[] = new int[5];
```

3.3 variables

Local variables should be initialized where they are declared if possible, while class variables are to be initialized in the class's constructor.

i.e.,

```
public boolean enterPin( int tryCtr ) {

        String inputPin; //cannot be initialized yet
        Scanner sc = new Scanner( System.in ); //can be initialized

        for( int i = 0; i < tryCtr; i++ ) {
                inputPin = sc.readLine();
                if( inputPin.equals( pin.getPin() ) ) { //assume pin is declared elsewhere in the class
                        return true;
                }
        }
        return false;
}
```

All variables should have only *one* meaning.

Class variables could be declared as *protected* if the class is intended to be a superclass, otherwise declare all class variables as *private*.

3.4 loops

Initialize iterators inside of the for loop

i.e.,

```
for (int i = 0; i < 100; i++)     // NOT:  int i;
   sum += addend[i];                       for (i= 0, sum = 0; i < 100; i++)
                                               sum += addend[i];
```

***Circumvent* the use of *do-while loops* unless absolutely necessary.**

3.5 conditionals

The conditional should always be put on a different line.

i.e.,

```
if (isAlive)       // NOT: if (isAlive) catchAnimon();
    catchAnimon();
```

Refrain from using executable statements in conditionals.

i.e., (Example is subject to change)

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
    :
}

// NOT:
if (File.open(fileName, "w") != null)) {
    :
}
```

3.6 Miscellaneous

Refrain from using arbitrary numbers; constants can be declared for use of numbers other than 0 and 1.

i.e., (Example is subject to change)

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];
```

The representation of floating-point numbers should always be with at least one decimal and with a decimal point.

i.e., (Example is subject to change)

```
double total = 0.0;    // NOT:  double total = 0;
double speed = 3.0e8;  // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

A digit must always be placed before the decimal point for floating-point numbers.
i.e., (Example is subject to change)

```
float total = 0.13;  // NOT:  float total = .13;
```

4 LAYOUTS AND COMMENTS

4.1 Layouts

White spaces should not be used for indentions. Use TAB instead.
i.e.,

```
for (i = 0; i < nElements; i++)
    a[i] = 0;
```

Block layout should be as illustrated in example 1.
i.e.,

```
while (!done) {
    doSomething();
    done = moreToDo();
}
```

```
Not:
while (!done)
{
    doSomething();
    done = moreToDo();
}
```

```
Not:
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

***if-else* statements should be in the following form:**
i.e.,

```
if (condition) {
    statements;
}

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

if (condition) {
    statements;
}
else if (condition) {
    statements;
}
else {
    statements;
}
//Can be written without braces if there's only one statement
```

for, ***while*** **and** ***do-while*** **statements should be in the following form:**
i.e.,

```
for (initialization; condition; update) { //Can be written without braces if there's only one
    statements;                              //statement
}

while (condition) {  //Can be written without braces if there's only one
    statements;       //statement
}

do {
    statements;
} while (condition);
```

switch **statements should have the following form:**
i.e.,

```
switch (condition) {
     case ABC : statements;
                  ....
                  statements;

       case DE : statements;
```

```
                    break;

  case XYZW : statements;
                    ....
                    statements;
                    break;

      default : statements;
                   break;
}
```

***try-catch* statements should have the following form:**

i.e.,

```
try {
   statements;
} catch (Exception exception) {
   statements;
}

try {
   statements;
} catch (Exception exception) {
   statements;
} finally {
   statements;
}
```

4.2 spacing

Space characters should exist:
- Before and after operators.
- Before and after colons.
- After commas.
- After semicolons in for statements.

i.e.,

```
a = (b + c) * d;  // NOT: a=(b+c)*d

doSomething(a, b, c, d);   // NOT: doSomething(a,b,c,d);

case 100 :   // NOT: case 100:

for (i = 0; i < 10; i++) {   // NOT: for(i=0;i<10;i++){
  ...
```

Methods should be separated by 2 blank lines.

i.e.,

```
public getMax(){
}


public getMin(){
}
```

4.3 comments

There should be a space after the comment identifier.

i.e.,

```
// This is a comment    NOT: //This is a comment
/**                     NOT: /**
 * This is a javadoc            *This is a javadoc
 * comment                      *comment
 */                              */
```

All comments should be written in English.

Logical units in a block should be separated by one blank line with a descriptive comment in the line above each grouping.

i.e.,

```
//Setting up first panel
JPanel buttonsPanel = new JPanel();
this.add(buttonsPanel);

// Setting up first button
JButton loadButton = new JButton(“Load”);
loadButton.setBounds(5, 5, 50, 20);
buttonsPanel.add(loadButton);

// Setting up second button
JButton saveButton = new JButton(“Save”);
saveButton.setBounds(5, 25, 50, 20);
buttonsPanel.add(saveButton);
```

Use // for non-Javadoc comments, including multi-line comments.

i.e.,

```
// A single-line comment

// A multi-line comment
//continuation of the multi-line comment

/**
 *Example of a Javadoc comment.
 *Example of a Javadoc comment
 *Example of a Javadoc comment
 */
```