276-2021-7-Iron: Java Code Style Guide

Make your code clean and simple

```
// Option 1:
final String HIGHEST_PRIORITY = "clean code";
System.out.println("My priority: " + HIGHEST_PRIORITY + ".");

// Option 2:
char[]w="\0rd0ct33lri".toCharArray();
for(int x=0xA;w[x]>0;System.out.print(w[x--]));
```

It is always best to write simple, clear code that is easy to understand, debug and maintain.

Use meaningful names

This includes variables, classes, functions, etc.

```
void display();
int countStudents();
Date dateStudentRegistered;
```

Exception 1: loop variables may be `i`, `j` or `k`. However, prefer the for-each loop when possible.

```
for (int i = 0; i < 10; i++) {
    ...
}
```

Exception 2: variables with very limited scope (<20 lines) may be shortened if the purpose of the variable is clear.

```
void swapCars(Person person1, Person person2)
{
    Car tmp = person1.getCar();
    person1.setCar(person2.getCar());
    person2.setCar(tmp);
}
```

Naming Conventions:

Constants must be all upper case, with multiple words separated by '_':

```
final int DAYS_PER_WEEK = 7;
```

Functions must start with a lower case letter, and use CamelCase. Functions should be named in terms of an action:

```
double calculateTax();
boolean verifyInput();
```

Class names must start with an upper case letter, and use CamelCase. Classes should be named in terms of a singular noun:

```
class Student;
class VeryLargeCar;
```

Note: Constants should have the most restrictive scope possible. For example, if it is used in only one class, then define the constant as private to that class. If a constant is needed in multiple classes, make it public.

Do not use prefixes for variables: don't encode the type (like iSomeNumber, or strName), do not prefix member variables of a class have with m_ (avoid m_ChildName, or mChildName)

```
class Car {
    private String make;
    private int serialNumber;
    ...
}
```

Boolean variables named so that they make sense in an if statement:

```
if (isOpen) {
    ...
}
while (!isEndOfFile && hasMoreData()) {
    ...
}
```

Use constants instead of literal numbers (magic numbers):

```
// OK:
int i = 0;
i = i + 1;

// Bad: What are 0 and 1 for?!?
someFunction(x, 0, 1);
```

Indentation and Braces {..}

Tab size is 4; indentation size is 4. Use tabs to indent code.

```
if (j < 10) {
→   counter = getStudentCount(lowIndex,
→   →   highIndex);
→   if (x == 0) {
→   →   if (y != 0) {
→   →   →   x = y;→   →   // Insightful comment here
→   →   }
→   }
}
```

Opening brace is at the end of the enclosing statement; closing brace is on its own line, lined up with the start of the opening enclosing statement. Statements inside the block are indented one tab.

```
for (int i = 0; i < 10; i++) {
→   ...
}

while (i > 0) {
→   ...
}

do {
→   ...
} while (x > 1);

if (y > 500) {
→   ...
} else if (y == 0) {
→   ...
} else {
→   ...
}
```

Exception: Starting brace can start on left if it makes it easier to see the code:

```
if (someBigBooleanExpression
→   && !someOtherExpression)
{
→   ...
}
```

All if statements and loops should include braces!

```
if (a < 1) {
    a = 1;
} else {
    a *= 2;
}
while (count > 0) {
    count--;
}
```

Statements and Spacing

Declare each variable in it's own line! Not like this: int p1, p2;

```
int *p1;
int p2;
```

Each statement should be on its own line:

```
// Good:
i = j + k;
l = m * 2;

// Bad (what are you hiding?):
if (i == j) l = m * 2;
    cout << "Can ya read this?" << endl;
```

All math operations must have spaces surrounding them. Commas should be followed by a space.

```
i = 2 + (j * 2) + -1 + k++;
if (i == 0 || j < 0 || !k) {
    arr[i] = i;
}
myObj.someFunction(i, j + 1);
```

Add extra brackets in complex expressions.

```
if ((!isReady && isBooting)
    || (x > 10)
    || (y == 0 && z < (x + 1)))
{
    ...
}
```

It is usually better to simplify complex expressions into multiple sub-expressions:

```
boolean isFinishedBooting = (isReady || !isBooting);
boolean hasTimedOut = (x > 10);
boolean isOldFirmware = (y == 0 && z < (x + 1));
if (!isFinishedBooting
    || hasTimedOut
    || isOldFirmware)
{
    ...
}
```

Classes

Inside a class, **all of the fields must be at the top of the class**, followed by the methods.

```
class Pizza {
    private int toppingCount;

    public Pizza() {
        toppingCount = 0;
    }
    public int getToppingCount() {
        return toppingCount;
    }
    ...
}

class Topping {
    private String name;
    ...
    public String getName() {
        return name;
    }
}
```

Comments

Use // for comments on one line.
Use // or /* .. */ for a couple lines.
Many lines should use /* .. */
Each class should have a JavaDoc format:

```
/**
 * Student class models the information about a
 * university student. Data includes student number,
 * name, and address. It supports reading in from a
 * file, and writing out to a file.
 */
class Student {
    ...
}
```

Function names should be obvious so we don’t need any comments inside the class.
If you do need comments describing something, always put it right before it.

Comments vs. Functions:

Your code should not need many comments. Generally, before writing a comment, consider how you can refactor your code to remove the need to "freshen" it up with a comment.

When you do write a comment, it must describe why something is done, not what it does:

```
// Cast to char to avoid runtime error for
// international characters when running on Windows
if (isAlpha((char)someLetter)) {
    ...
}
```

Other:

Either post-increment or pre-increment may be used on its own:

```
i++;
++j;
```

Avoid using goto. When clear, design your loops to not require the use of `break` or `continue`.

All switch statements should include a `default` label. If the `default` case seems impossible, place an `assert false;` in it. Comment any intentional fall-throughs in `switch` statements:

```
switch(buttonChoice) {
case YES:
    // Fall through
case OK:
    System.out.println("It's all good.");
    break;
case CANCEL:
    System.out.println("It's over!");
    break;
default:
    assert false;
}
```

Use plenty of assertions. Any time you can use an assertion to check that some condition is true which "must" be true, you can catch a bug early in development. It is especially useful to verify function pre-conditions for input arguments or the object's state. Note that you must give the JVM the `-ea` argument (enable assertions) for it to correctly give an error message when an assertion fails.

Never let an assert have a side effect such as `assert i++ > 1;`. This may do what you expect during debugging, but when you build a release version, the asserts are removed. Therefore, the i++ won't happen in the release build.