Java Code Style Guide

Clean & Simple Code

Which option are you?

```
// 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

All identifiers must have meaningful, human-readable, English names. Avoid cryptic abbreviations such as `dspl()`, `cntStd()`, or `stdRegYYYYMMDD`. Instead use:

```
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;
```

Constant 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.

Generally favour local variables over "more global" variables such as class fields. In almost all languages, global variables are terrible!

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_`.

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

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

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

Use named constants instead of literal numbers (magic numbers). It is often acceptable to use 0 and 1; however, it must be clear what they mean:

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

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

Indentation and Braces {...}

There's **always** time for **perfect indentation**.

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: `if` statements with multi-line conditions have the starting brace aligned on the left to make it easier to spot the block in the `if` statement.

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

All `if` statements and loops should include braces around their statements, even if there is only one statement in the body:

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

Your indenation may differ in cases where it improves readability. For example with a multi-line array, complex conditionals or complex calculations.

Statements and Spacing

Declare each variable in its own definition, rather than together (`int i, j`).

```
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 binary (2 argument) operators (arithmetic, bitwise and assignment) and ternary conditionals (?:) must be surrounded by one space. Commas must have one space after them and none before. Unary operators (!, *, &, - (ex: -1), + (ex: +1), ++, --) have no additional space on either side of the operator.

```
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, even if operator precedence will do what you want. The extra brackets increase readability and reduce errors.

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

However, it is often better to simplify complex expressions by breaking them into multiple sub-expressions that are easier to understand and maintain:

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

```
    ...
}
```

Classes

Inside a class, 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;
    }
}
```

Enums

Prefer the use of enums over integer constants if you indicating some specific things, such as north/south/east/west directions.

```
class Money {
    public enum Coin {
        PENNY,
        NICKLE,
        DIME,
        QUARTER,
```

```
        LOONIE
    }

    private List coins = ...;
    ...
    public void addCoin(Coin coin) {
        if (coin == Coin.PENNY) {
            System.out.println("Found a penny!");
        }
        coins.add(coin);
    }
}
```

Comments

Comments which are on one line should use the `//` style. Comments which are or a couple lines may use either the `//`, or `/* ... */` style. Comments which are many lines long should use `/* ... */`.

Each class must have a descriptive comment before it describing the general purpose of the class. These comments should be in the JavaDoc format. Recommended format is shown below:

```
/**
 * 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 {
    ...
}
```

Each member function in a class may have a comment describing what it does. However, the name of the function, combined with an understanding of the class, should be enough to tell the user what it does.

Comments should almost always be the line before what they describe, and be placed at the same level of indentation as the code. Only very short comments may appear inline with the code:

```
// Display final confirmation message box.
callSomeFunction(
        0,              // Parent.
        "My App",       // Title
        "test");        // Message
```

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.