George Fox University CS/IS/Cyber Java Coding Style Guidelines

Introduction

- 3 or 4 spaces are used as your default indentation… DO NOT use tabs.
- Local variable (including parameters) and method names are lowercase, with occasional `upperCase` characters in the middle. Use `camelCase`
- Class names start with an `Uppercase` letter
- Constant names are `UPPERCASE`, with an occasional `UNDER_SCORE`.
- There are spaces after reserved words and surrounding binary operators, i.e.
  - `total = num1 + num2;  //Good`
  - `total=num1+num2;  //Poor`
  - `if (someVar < someValue)  //Good`
  - `if(someVar<someValue)  //Poor`
  - `for (…)  //Good`
  - `for(…)  //Poor`
- Braces must line up vertically.
- No magic numbers may be used (use variables or constants).
- Every method must have a Javadoc header comment.
- At most 30 lines of code may be used per method.
- No continue or break is allowed.
- All non-final instance variables must be private and start with an underscore then lowercase letter.

Classes

Each class should be preceded by a Javadoc class comment explaining the purpose of the class.

Use the following ordering when creating your class:

1. Static final variables
2. Static variables
3. Instance variables
4. Constructors
5. Instance methods
6. Static methods
7. Main (if present)

Leave 2 blank lines after every method.

All non-final variables must be `private`. Methods and final variables can be either `public` or `private`, as appropriate.

All features must be tagged `public` or `private` (or `protected` when appropriate).

Methods

Every method (except for main) starts with a comment in Javadoc format.

```
/**
   Convert calendar date into Julian day.
   Note: This algorithm is from Press et al., Numerical Recipes
   in C, 2nd ed., Cambridge University Press, 1992

   @param day day of the date to be converted
   @param month month of the date to be converted
   @param year year of the date to be converted
   @return the Julian day number that begins at noon of the
   given calendar date.
   [optional]@throws as necessary
*/
public static int getJulianDayNumber(int day, int month, int year)
{
   . . .
}
```

Parameter names must be explicit, especially if they are integers or Boolean:

```
public Employee remove(int d, double s)
   //  Huh?
public Employee remove(int department, double severancePay)
   //  OK
```

Methods must have at most 30 lines of code. The method signature, comments, blank lines, and lines containing only braces are not included in this count. This rule forces you to break up complex computations into separate methods.

Variables and Constants

Define all variables at the beginning of methods:

```
public void whatever()
{
   double xold;
   double xnew;
   boolean done;
   . . .
}
```

Do not define each variable just before it is used for the first time ("On The Fly") `xnew` is declared 'on the fly' below:

```
{
   . . .
   double xold = Integer.parseInt(input);
   boolean done = false;
   while (!done)
   {
      double xnew = (xold + a / xold) / 2;
      . . .
   }
   . . .
}
```

Do not define multiple variables on the same line:

```
int dimes = 0, nickels = 0; // Don't
```

Instead, use separate definitions:

```
int dimes = 0;      // OK
int nickels = 0;
```

In Java, constants must be defined with the reserved word final. If the constant is used by multiple methods, declare it as static final. It is a good idea to define static final variables as private if no other class has an interest in them.

Do not use magic numbers! A magic number is a numeric constant embedded in code, without a constant definition. Any number except -1, 0, 1, and possibly 2 is considered magic:

```
if (p.getX() < 300) // Don't
```

Use final variables instead:

```
final double WINDOW_WIDTH = 300;
. . .
if (p.getX() < WINDOW_WIDTH) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
public static final int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

When declaring array variables, group the [] with the type, not the variable.

```
int[] values; // OK
int values[]; // Ugh--this is an ugly holdover from C
```

Control Flow

The `if` Statement

Avoid the "if . . . if . . . else" trap. The code

```
if ( . . . )
   if ( . . . ) . . .;
else . . .;
```

will not do what the indentation level suggests, and it can take hours to find such a bug.

Always use { . . . }'s when dealing with "if . . . if . . . else":

```
if ( . . . )
{
   if ( . . . )
   {
      . . .;
   }
   else
   {
      . . .;
   }
}
```

The `for` Statement

Use for loops only when a variable runs from one value to another with some constant increment/decrement:

```
for (int i = 0; i < nums.length; i++)
   System.out.println(nums[i]);
```

Or, even better, use the "for each" loop:

```
for (int num : nums)
   System.out.println(num);
```

Do not use the for loop for weird constructs such as

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew))
   // Don't
```

Make such a loop into a while loop. That way, the sequence of instructions is much clearer.

```
a = a / 2;
while (count < ITERATIONS) // OK
{  . . .
   System.out.println(xnew);
}
```

Nonlinear Control Flow

Avoid the break or continue statements. Use another `boolean` variable to control the execution flow.

Exceptions

Do not tag a method with an overly general exception specification:

```
Widget readWidget(Reader in)
   throws Exception // Bad
```

Instead, specifically declare any checked exceptions that your method may throw:

```
 Widget readWidget(Reader in)
    throws IOException, MalformedWidgetException // Good
```

Do not "squelch" exceptions:

```
try
{
    double price = in.readDouble();
}
catch (Exception e)
{} // Bad
```

Beginners often make this mistake "to keep the compiler happy". If the current method is not appropriate for handling the exception, simply use a throws specification and let one of its callers handle it.

Lexical Issues

Naming Convention

The following rules specify when to use upper- and lowercase letters in identifier names.

All local variable and method names are in lowercase (maybe with an occasional upperCase in the middle); for example, `firstPlayer`. Private instance variables use the same naming convention except they are preceded by an underscore; for example `_totalCapacity`.

All constants are in uppercase (maybe with an occasional `UNDER_SCORE`); for example, `CLOCK_RADIUS`.

All class and interface names start with uppercase and are followed by lowercase letters (maybe with an occasional UpperCase letter); for example, `BankTeller`.

Names must be reasonably long and descriptive. Use `firstPlayer` instead of `fp`. No drppng vwls. Local variables that are fairly routine can be short (ch, i) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use ctr, c, cntr, cnt, c2 for variables in your method. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, `current`, `next`, `previous`, `result`, . . .).

Indentation and White Space

Use 3 or 4 spaces for your indentation. (pick one and be consistent)

Use blank lines freely to separate parts of a method that are logically distinct.

Use a blank space around every binary operator:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);  // Good


x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);//Bad
```

Leave a blank space after (and not before) each comma or semicolon. Do not leave a space before or after a parenthesis or bracket in an expression. Leave spaces around the (. . .) part of an if, while, for, or catch statement.

```
if (x == 0)
    y = 0;


f(a, b[i]);
```

Every line must fit on 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = ..................................................
   + .................;
```

Start the indented line with an operator (if possible).

If the condition in an if or while statement must be broken, be sure to brace the body in*, even if it consists of only one statement*:

```
if ( ..................................................
       && ..................
       || .......... )
{
   . . .
}
```

If it weren't for the braces, it would be hard to separate the continuation of the condition visually from the statement to be executed.

Braces

Opening and closing braces must line up vertically:

```
while (i < n)
{
   System.out.println(a[i]);
   i++;
}
```

Some programmers don't line up vertical braces but place the { behind the reserved word:

```
while (i < n) {     // DON'T
   System.out.println(a[i]);
   i++;
}
```

Doing so makes it hard to check that the braces match.

<u>NOTE: If you have questions on style that are not covered in this document… ASK (before you submit your work)</u>