Untitled Solutions Coding Style Guide

This guide was constructed to have a consistent coding style between all the members of Untitled Solutions. While much of this guide was written by hand, many of the rules and examples shown will originate from other documents written by organisations with far more experience than our own. These documents will be listed at the end of this document but will not be referenced throughout.

Terminology

1. ‘Columns’ refer to the space that a single character takes up within a line.
    1.1. The expression “if” for example takes up two columns, a character always takes up a column regardless of which character it is. A “Column Limit” should be the limit of characters within a line.
2. Indent refers to the amount of whitespace before a statement, for example statements within a method or the content of an if statement are often indented for readability purposes.

Formatting

1. Column Limit is set at 100.
2. Indentation is four columns for methods, classes, if, for and while statements.
    2.1. This is the default for Intellij and it should do this for you automatically.
3. Variables and method names should be written in camelCase, class names should be written in PascalCase and constants or final variables should be written in UPPER_SNAKE_CASE.
    3.1. If you’re unfamiliar with these cases, they have been written in their format as a reference.
4. All method and variable names should clearly reflect their purpose, a goal when coding should be that most, ideally all, of your code is easily decipherable even without the comments.
    4.1. The only variables that do not require meaningful names are the iterators within for loops, due to their purpose often being clear even without being named; foreach loops however, do require meaningful names.
    4.2. Temporary variables will also require names more meaningful than simply “temp”, with some description as to what they are temporary variables of.
5. Our indentation style is the “One True Brace Style”:

```
// Like this.
if (x < 0) {
  negative(x);
} else {
  notNegative(x);
}

public void method() {
}

// Not like this.
if (x < 0)
  negative(x);
else
```

```
  notNegative(x);

if (x < 0)
{
  negative(x);
}
else
{
  notNegative(x);
}

public void method()
{
}
```

6. Operators should be spaced and separated by explicit parenthesis where appropriate.

```
// Bad.

Integer foo=5*4+3/9^2;

// Good.

Integer foo = (5 * 4) + (3 / (9 ^ 2));
```

Documentation

1. Classes should have a comment above them with a short description of their purpose including the authors of a file, which should be all the people who have worked on that file and the date the file was last altered.
2. All methods should have a brief but descriptive summary of their function and purpose in a Javadoc style. If a method cannot be described briefly this points to low cohesion of the method and that it may need restructuring.
3. Currently we only require that a summary of the method is provided and not the extraneous features of a Javadoc. For example, parameters and return types are not required, unless the return types or parameters are themselves vague.
4. “TODO: “s should be added to all code that is only partially functional, has some edge error cases that need to be resolved, or needs to be restructured in some way; as a reminder.
5. ALL comments should be pertinent and USEFUL, no comments should be added as “jokes”, signs of frustration or using a shorthand that only you understand. Comments are not written solely for the eyes of the commenter, they are written for everyone.
6. Any code written that may not be easily understood at a glance, which follows our other guidelines, should have comments explaining any actions that might be ambiguous to anyone reading them.

Imports

1. No wildcard imports, if you are using a specific data structure or object which requires importing, be explicit in what you are using.

```
// Bad.

Import java.util.*;
```

```
// Good.

Import java.util.ArrayList;
```

1.1. This is mainly done so thought is exercised when using datatypes to lower miscellaneous types being used un-necessarily but also improves clarity for anyone reviewing code later down the line.

Testing

1. Tests should be planned for all classes before the classes themselves are written.
   1.1. Through experience we have found that classes are both more likely to have low cohesion and be difficult for test if a test plan is not constructed for a specific class before coding on it has begun.
2. Tests should cover the expected states of the class, to make sure it functions well in all circumstances.
3. Tests should not include random elements, adding randomness to a test makes the results of the test unreliable, which adds difficulty to finding the fault within the program.
4. Tests should initialise their own variables and not be dependent on pre-established ones.
   4.1. For example, if a test requires a user with certain permissions to be run, that user should be generated and finally removed by the tests, instead of the tests relying on a pre-existing user which may be removed.
   4.2. Classes should ideally also be written in a way where these variables can be easily instantiated by the test methods and not be so procedural as to disallow it.

General Principles

1. We are using a DRY (Don't Repeat Yourself) methodology, so an attempt should be made to make extremely similar methods as more generalised methods which can handle either input, rather than making several methods which all do slightly different things.
   1.1. This excludes the case where code complexity would rise significantly when attempting to achieve the generalised function.
2. Unambiguous statements and code should be written where possible, operators using explicit parenthesis is an extension of this rule, but this applies to anything that may lower readability.

```
// Bad, may be seen as 1001 or 100L.
long count = 100l + n;

// Good.
long count = 100L + n;
```

3. Classes with methods and variables that require access from within the package, but should not otherwise be public, will be prefaced with no modifier, making them package-private.
   3.1. This is done because we wish other classes in the package to have access to that method, but not any classes from outside our package.
4. Delete unused or deprecated code.

5. Don’t use temporary variables when a single method call will do.

```
// Bad, assignment without purpose.
String name = Person.getName();
return name;

// Good.
return Person.getName();
```

6. Typecasting is seen as a logical error and should be avoided where possible.
7. Avoid using null where possible.
   7.1. Where it should be used it should be clearly signposted and discussed with the group.
8. Use primitive types unless methods requiring the class implementation are used, or if a reference to the data is required instead of the data itself.
9. A blank line should be placed between methods of a class for clarity.
10. Where possible, logically distinct lines of code in a method should be separated by an empty line and a single-line comment describing the operation being done.
11. Logically similar methods should be placed near each other if it is reasonably easy to do so.

References:

Twitter’s Java Style Guide - https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md