CIS 120: Programming Languages and Techniques I Spring 2017

CIS 120 Java Style Guide

Contents

Formatting

(FL) 100-character line length: Ensure that no lines of code are longer than the maximum number of characters; in the case of CIS 120 Java programming, this limit is 100 characters due to Java's inherent verbosity.

(FI) Indentation using spaces: Do not indent with tab characters (`\t`). Any block of code enclosed in curly braces should be indented one level deeper than the surrounding code. Choose a reasonable and consistent convention for indentation; generally 2 to 4 spaces is acceptable. CIS 120 recommends 4 spaces for indentation.

(FC) Use curly braces consistently: Using either "Egyptian" (images/egyptian_braces.jpg) curly braces or C-style curly braces is acceptable, as long as the choice is *consistent* throughout the program. CIS 120 and the official Java style guidelines recommend the former.

```
/* Egyptian braces -- Preferred */
if (a == b) {
    System.out.println("these are Egyptian braces");
} else {
    System.out.println("hello, world!");
}

/* C-style braces -- Acceptable */
if (a == b)
{
    System.out.println("these are C-style braces");
}
else
{
    System.out.println("hello, world!");
}
```

(FB) Curly braces around blocks: Though Java permits the elision of curly braces in cases where the body `if`-statement or loop consists of a single statement, we require that every block (no matter the number of statements) be enclosed inside curly braces on a new line. It is far too easy to make programming errors of this form otherwise:

```
// NO
while (x < 5)
    System.out.println("Inside block");
    System.out.println("BUG -- Not inside block!");
```

(FS) One statement per line: There should be no more than one statement (declaration, assignment, or function call) on any line of your program.

(FW) Use vertical whitespace to separate chunks of code: Within a block of code, use vertical whitespace (blank lines) to separate groups of statements. This makes code more readable and clarifies which statements are logically related.

(FO) Always put spaces around operators: Every Java operator should have spaces around it. Use parentheses to communicate precedence. For example: `5 - (x + 4) * 8`

Naming

(NM) Naming variables and methods: Use lowerCamelCase for variable and method names. Single-word variables should be all lowercase; subsequent words should be joined with their first character capitalized. These identifiers should not contain underscores.

(NC) Naming classes: Use UpperCamelCase for class names and enum type names. Do not capitalize acronyms within camelCased names; the string `"TCP socket ID"` should be written `TcpSocketId` rather than `TCPSocketID`.

(NV) Naming constant and enum values: Values that are constants, including final static variables and enum values, should follow CAPITAL_CASE conventions, in which all-caps words are separated by underscore characters.

(NS) More descriptive names for larger scopes: Variables that have greater scope should have more descriptive names. It is often preferred to use a short name like `i` or `j` for a loop index, but as the scope of an identifier increases, so should the meaningfulness of its name. For example, prefer `leftChild` over `l` for a class field.

Commenting

(CO) Do not over-comment code: If the function a piece of code is reasonably obvious to experienced Java programmers, then it likely does not need to be commented. Be judicious; if you are unsure, it is often best not to include a redundant comment. If you feel like it is necessary to extensively comment your code, consider how to rewrite it to make it more straightforward.

(CF) Comments should describe purpose over implementation: The code itself is a description of an implementation, so it is unnecessary to comment what the code is doing (e.g., `x++` does not need a comment like "Increment x"). Instead, describe at a high level the intended use of the piece of code, such as what task the function performs.

(CD) Write Javadoc comments: Above all public methods and non-trivial private methods, you should include a block comment with the Javadoc syntax `/** ... */`. This should describe the function's intended use, as well as information about the function's parameters, return value, and exceptions it throws (if any).

(CC) Do not submit commented-out code: Remove any dead or commented-out code from your source files before submission. If you need to save snapshots of code, consider using a separate file or placing your homework under version control.

Verbosity

(VH) Use a helper variables or methods to avoid computing values multiple times: If you find that a particular sequence of computations is being repeated more than twice, you should assign it to a variable and/or abstract it into a helper function which returns the desired value.

(VE) Avoid redundant or verbose expressions: Write expressions such as `if` conditions, `while` loop guards, and the like in the most succinct and expressive way possible. Especially consider whether a long expression involving multiple `||` or `&&` operators is easily readable and whether it could be written with fewer or simpler sub-expressions.

(VI) Prefer Java idioms over more verbose alternatives: There are often many ways to write the same code in Java; make use of language features and idioms that make code more compact and easier to understand at a glance. For example, consider the following loop, which prints the contents of the list `l`:

```
List<String> l = /* ... */
for (int i = 0; i < l.size(); i++) {
    System.out.println(l.get(i));
}
```

Contrast this with a more idiomatic version that uses the for-each construct:

```
List<String> l = /* ... */
for (String s : l) {
    System.out.println(s);
}
```

The second version has less syntatctic clutter, and its intent is much more obvious to the reader of the code. (It is also much faster if `l` is a `LinkedList`!)

(VL) Make use of library functions when possible: Java has quite a rich standard library, so you should make use provided functions instead of re-implementing them. (Your implementation will almost always be less efficient, and is less likely to be correct.)

Organization

(OM) Ordering of class methods and values: Classes should be organized internally according to this order:

1. Import statements
2. Fields and state variables
3. Constructors
4. Methods

(OC) Methods and classes should perform one specific and unique task: Avoid creating a single method or class that "knows too much"; that is, a method or class which serves multiple disjoint purposes or is responsible for too much. Such implementations are often unwieldy and difficult to extend or debug. This is known as the principle of separation of concerns—each class and method should be responsible for one thing and one thing only. Your program should be composed of a group of such classes which cooperate to achieve a goal.

(OE) Use extension only to represent "is-a" relationships: If you create a subclass, make sure that the subtype `A` "is a" version or variant of the supertype `B`. If it is more appropriately described as a "has-a" relationship, then consider making `B` a field of `A` instead.

Data Structures

(DS) Consider using `Set`s to represent unordered data: The abstract type `Set` is a good (though not always the best) candidate for representing data that does not require any particular ordering. `Set`s also provide efficient verification of object membership. Example implementation in Java: `TreeSet`.

(DL) Consider using `List`s to represent ordered data: The abstract type `List` is a good (though not always the best) candidate for representing data with an imposed or sorted ordering. `List`s generally provide efficient insertion of elements at their endpoints. Example implementation in Java: `LinkedList`.

(DM) Consider using Maps to represent associations between data types: The abstract type `Map` is a good (though not always the best) candidate for representing one-to-one associations or relationships between two objects. `Map`s, like `Set`s, provide efficient verification of object membership, as well as efficient value lookup for a given key.

(DT) Static types should be interfaces: The declared static type of any collection should be a Java interface rather than an implementation of that interface. For example:

```
Map<String, Integer> ids = new TreeMap<>(); // Map, not TreeMap
List<Point> points = new LinkedList<>();    // List, not LinkedList
```

JUnit Testing

(TX) Tests expecting exceptions: If you are writing a test case that is expected to throw an exception, you should add an expected value to the `@Test` annotation. For example, if we expect the body of our test function to throw an `IllegalArgumentException`, we would write `@Test(expected=IllegalArgumentException.class)` above the function. It will fail if it does not raise an instance of this particular exception. Do not use `try-catch` statements in place of the expected parameter.

(TE) Use `assertEquals(a,b)` over `assertTrue(a.equals(b))`: JUnit provides comparison assertions (which implicitly call objects' `equals` methods), but provide more useful failure information than `assertTrue`.

(TF) Test floating-point values with a threshold: Do not use JUnit's default `assertEquals(double expected, double actual)` method, as it is deprecated. Instead, use `assertEquals(double expected, double actual, double epsilon)`, which allows for some wiggle room (an `epsilon` value) to account for the imprecision of floating-point values. An `epsilon` value of 0.0001 usually suffices for most purposes.